The Seasoned Schemer

Daniel P. Friedman and Matthias Felleisen

Foreword and Afterword by Guy L. Steele Jr.
The First Ten Commandments

The First Commandment
When recurring on a list of atoms, \( \text{lat} \), ask two questions about it: \( \text{null? lat} \) and else.
When recurring on a number, \( n \), ask two questions about it: \( \text{zero? n} \) and else.
When recurring on a list of S-expressions, \( l \), ask three question about it: \( \text{null? l}, \text{atom? (car l)}, \text{and else.} \)

The Second Commandment
Use \text{cons} to build lists.

The Third Commandment
When building a list, describe the first typical element, and then \text{cons} it onto the natural recursion.

The Fourth Commandment
Always change at least one argument while recurring. When recurring on a list of atoms, \( \text{lat} \), use \( \text{cdr lat} \). When recurring on a number, \( n \), use \( \text{subl n} \). And when recurring on a list of S-expressions, \( l \), use \( \text{car l} \) and \( \text{cdr l} \) if neither \( \text{null? l} \) nor \( \text{atom? (car l)} \) are true.
It must be changed to be closer to termination. The changing argument must be tested in the termination condition:
when using \text{cdr}, test termination with \text{null?}
and
when using \text{subl}, test termination with \text{zero?}.

The Fifth Commandment
When building a value with \( + \), always use 0 for the value of the terminating line, for adding 0 does not change the value of an addition.
When building a value with \( \times \), always use 1 for the value of the terminating line, for multiplying by 1 does not change the value of a multiplication.
When building a value with \text{cons}, always consider () for the value of the terminating line.

The Sixth Commandment
Simplify only after the function is correct.

The Seventh Commandment
Recur on the \text{subparts} that are of the same nature:
- On the sublists of a list.
- On the subexpressions of an arithmetic expression.

The Eighth Commandment
Use help functions to abstract from representations.

The Ninth Commandment
Abstract common patterns with a new function.

The Tenth Commandment
Build functions to collect more than one value at a time.
The Eleven Commandment
Use additional arguments when a function needs to know what other arguments to the function have been like so far.

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

The Thirteenth Commandment
Use (letrec ...) to hide and to protect functions.

The Fourteenth Commandment
Use (letcc ...) to return values abruptly and promptly.

The Fifteenth Commandment
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. And use (let ...) to name the values of expressions (without set!) that are re-evaluated every time a function is used.

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

The Seventeenth Commandment
Use (set! x ...) for (let ((x ...)) ...) only if there is at least one (lambda ...) between it and the (let ...), or if the new value for x is a function that refers to x.

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

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

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.
The Seasoned Schemer

Daniel P. Friedman
Indiana University
Bloomington, Indiana

Matthias Felleisen
Rice University
Houston, Texas

Drawings by Duane Bibby

Foreword and Afterword by Guy L. Steele Jr.

The MIT Press
Cambridge, Massachusetts
London, England
To Mary, Helga, and our children
((Contents)

(Foreword ix)
(Preface xi)
((11. Welcome Back to the Show) 2)
((12. Take Cover) 16)
((13. Hop, Skip, and Jump) 36)
((14. Let There Be Names) 62)
((15. The Difference Between Men and Boys ...) 90)
((16. Ready, Set, Bang!) 106)
((17. We Change, Therefore We Are!) 126)
((18. We Change, Therefore We Are the Same!) 142)
((19. Absconding with the Jewels) 154)
((20. What's in Store?) 178)
(Welcome to the Show 204)
(Afterword 207)
(Index 209))
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 \texttt{atom?}, \texttt{sub1}, and \texttt{add1}, which we introduced in \textit{The Little Schemer}:

\begin{verbatim}
(define atom? (lambda (x) (and (not (pair? x)) (not (null? x)))))
\end{verbatim}

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

\begin{verbatim}
(defun atom? (x) (not (listp x)))
\end{verbatim}

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 \textit{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 \textit{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 \textit{italic}. Basic data, including numbers and representations of truth and falsehood, is set in \textit{sans serif}. Keywords, i.e., \texttt{letrec}, \texttt{letcc}, \texttt{let}, \texttt{if}, \texttt{set!}, \texttt{define}, \texttt{lambda}, \texttt{cond}, \texttt{else}, \texttt{and}, \texttt{or}, and \texttt{quote} are in \textbf{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 \texttt{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.

Have you read *The Little LISP* Per?\(^1\) 

- \(\text{#f.}\)

---

1 Or *The Little Scheme*.

---

Are you sure you haven’t read *The Little LISP* Per?

- Well, …

---

Do you know about Lambda the Ultimate?

- \(#t\).

---

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

- Absolutely.\(^1\)

---

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

---

Are you acquainted with *member*?

- Sure, *member*? is a good friend.

---

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

---

What is the value of \((\text{member}? \ a \ \text{lat})\) where \(a\) is sardines and 

\(\text{lat}\) is (Italian sardines spaghetti parsley)

- \(#t\), but this is not interesting.

---

What is the value of \((\text{two-in-a-row}? \ \text{lat})\) where 

\(\text{lat}\) 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.

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

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

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`?

```scheme
(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?

```scheme
(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?`.


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?

\[
\text{(define two-in-a-row-b?}
\begin{array}{l}
\quad \text{(lambda} \ (\text{preceding lat}) \\
\quad \quad \text{(cond} \\
\quad \quad \quad \text{((null? lat) #f)} \\
\quad \quad \quad \text{(else} ... \\
\quad \quad \quad \quad \text{(two-in-a-row-b?} \ (\text{car lat}) \\
\quad \quad \quad \quad \quad \text{(cdr lat)}) \\
\quad \quad \quad \quad \text{...)))})
\end{array}
\]

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

\[
\text{(define two-in-a-row-b?}
\begin{array}{l}
\quad \text{(lambda} \ (\text{preceding lat}) \\
\quad \quad \text{(cond} \\
\quad \quad \quad \text{((null? lat) #f)} \\
\quad \quad \quad \text{(else} \ (\text{or} \ (\text{eq?} \ (\text{car lat}) \ \text{preceding}) \\
\quad \quad \quad \quad \text{(two-in-a-row-b?} \ (\text{car lat}) \\
\quad \quad \quad \quad \quad \text{(cdr lat)))}))
\end{array}
\]

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

The natural recursion is

\[
\text{(two-in-a-row-b? (car lat) (cdr lat)).}
\]

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:

\[
\text{(define two-in-a-row?}
\begin{array}{l}
\quad \text{(lambda} \ (\text{lat}) \\
\quad \quad \text{(cond} \\
\quad \quad \quad \text{((null? lat) #f)} \\
\quad \quad \quad \text{(else} \ (\text{two-in-a-row-b?} \ (\text{car lat}) \\
\quad \quad \quad \quad \quad \text{(cdr lat))}))
\end{array}
\]

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

Okay.

Welcome Back to the Show
(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.
<table>
<thead>
<tr>
<th>Then how about the second line?</th>
<th>The second line is the hard part.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why?</td>
<td>The answer should be the sum of all the numbers that we have seen so far <em>consed</em> onto the natural recursion.</td>
</tr>
<tr>
<td>Let's do it!</td>
<td>The function does not know what all these numbers are. So we can't form the sum of the prefix.</td>
</tr>
<tr>
<td>How do we get around this?</td>
<td>The trick that we just saw should help.</td>
</tr>
<tr>
<td>Which trick?</td>
<td>Well, <em>two-in-a-row-b</em>? receives two arguments and one tells it something about the other.</td>
</tr>
<tr>
<td>What does <em>two-in-a-row-b</em>?s first argument say about the second argument.</td>
<td>Easy: the first argument, <em>preceding</em>, always occurs just before the second argument, <em>lat</em>, in the original list.</td>
</tr>
<tr>
<td>So how does this help us with <em>sum-of-prefixes</em></td>
<td>We could define <em>sum-of-prefixes-b</em>, which receives <em>tup</em> and the sum of all the numbers that precede <em>tup</em> in the <em>tup</em> that <em>sum-of-prefixes</em> received.</td>
</tr>
<tr>
<td>Let's do it!</td>
<td></td>
</tr>
</tbody>
</table>
|                                | (define *sum-of-prefixes-b*  
|                                |   (lambda (sonssf tup)  
|                                |     (cond  
|                                |       ((null? tup) (quote ()))  
|                                |       (else (cons (+ sonssf (car tup))  
|                                |         (sum-of-prefixes-b  
|                                |           (+ 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 
\((\text{sum-of-prefixes-b} \ \text{sonssf} \ \text{tup})\)
where \(\text{sonssf} = 0\)
and
\(\text{tup} = (1 \ 1 \ 1)\)

Since \(\text{tup}\) is not empty, we need to determine the value of
\((\text{cons} \ 1 \ (\text{sum-of-prefixes-b} \ 1 \ \text{tup}))\)
where
\(\text{tup} = (1 \ 1)\).

And what do we do now?

We \text{cons} 2 onto the value of
\((\text{sum-of-prefixes-b} \ 2 \ \text{tup})\)
where
\(\text{tup} = (1)\).

Next?

We need to remember to \text{cons} the value 3 onto \((\text{sum-of-prefixes-b} \ 3 \ \text{tup})\)
where
\(\text{tup} = ()\).

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 \text{sonssf} a good name?

Yes, every natural recursion with \text{sum-of-prefixes-b} uses the sum of all the numbers preceding \text{tup}.

Define \text{sum-of-prefixes} using \text{sum-of-prefixes-b}

Obviously the first sum for \text{sonssf} must be 0:

\begin{verbatim}
(define sum-of-prefixes
  (lambda (tup)
    (sum-of-prefixes-b 0 tup)))
\end{verbatim}

The Eleventh Commandment

Use additional arguments when a function needs to know what other arguments to the function have been like so far.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you remember what a tup is?</td>
<td>A tup is a list of numbers.</td>
</tr>
<tr>
<td>Is ((1\ 1\ 1\ 3\ 4\ 2\ 1\ 1\ 9\ 2)) a tup?</td>
<td>Yes, because it is a list of numbers.</td>
</tr>
<tr>
<td>What is the value of ((\text{scramble} \ tup)) where (tup = (1\ 1\ 1\ 3\ 4\ 2\ 1\ 1\ 9\ 2))?</td>
<td>((1\ 1\ 1\ 1\ 4\ 1\ 1\ 1\ 9)).</td>
</tr>
<tr>
<td>((\text{scramble} \ tup)) where (tup = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9))</td>
<td>((1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)).</td>
</tr>
<tr>
<td>((\text{scramble} \ tup)) where (tup = (1\ 2\ 3\ 1\ 2\ 3\ 4\ 1\ 8\ 2\ 10))</td>
<td>((1\ 1\ 1\ 1\ 1\ 1\ 1\ 2\ 8\ 2)).</td>
</tr>
<tr>
<td>Have you figured out what it does yet?</td>
<td>It’s okay if you haven’t. It’s kind of crazy. Here’s our explanation:</td>
</tr>
<tr>
<td></td>
<td>“The function \textit{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.”</td>
</tr>
<tr>
<td>If (l = (1\ 1\ 1\ 3\ 4\ 2\ 1\ 1\ 9\ 2)) what is the prefix of ((4\ 2\ 1\ 1\ 9\ 2)) in (l)</td>
<td>((1\ 1\ 1\ 3\ 4)), because the prefix contains the first element, too.</td>
</tr>
<tr>
<td>And if (l = (1\ 1\ 1\ 3\ 4\ 2\ 1\ 1\ 9\ 2)) what is the prefix of ((2\ 1\ 1\ 9\ 2)) in (l)</td>
<td>((1\ 1\ 1\ 3\ 4\ 2)).</td>
</tr>
</tbody>
</table>
Is it true that \((\text{scramble } \text{tup})\) must know something about the prefix for every element of \(\text{tup}\)?

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

Does this mean we have to define another function that does most of the work for \(\text{scramble}\)?

Yes, because \(\text{scramble}\) needs to collect information about the prefix of each element in the same manner as \(\text{sum-of-prefixes}\).

What is the difference between \(\text{scramble}\) and \(\text{sum-of-prefixes}\)?

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

What is \(\text{(pick } n \text{ lat)}\) where \(n\) is 4 and \(\text{lat}\) is \(\langle 4 \ 3 \ 1 \ 1 \ 1 \rangle\)?

1.

What is \(\text{(pick } n \text{ lat)}\) where \(n\) is 2 and \(\text{lat}\) is \(\langle 2 \ 4 \ 3 \ 1 \ 1 \ 1 \rangle\)?

4.

Do you remember \(\text{pick}\) from chapter 4?

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

\[
\text{(define pick (lambda } (n \text{ lat)) (cond \ ((\text{one? } n) (\text{car lat})) \ (else (pick (sub1 } n) (\text{cdr lat})))))}
\]
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)))))))
```

A better question is: how does it work?

How do we get `scramble-b` started?

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 consing `(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 `conses` 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 \((\text{multirember } a \ \text{lat})\)

where \(a\) is tuna

and

\(\text{lat}\) is \(\text{(shrimp salad tuna salad and tuna)}\)

(shrimp salad salad and),

but we already knew that from chapter 3.

Does \(a\) change as \(\text{multirember}\) traverses \(\text{lat}\)

No, \(a\) always stands for tuna.

Well, wouldn't it be better if we did not have to remind \(\text{multirember}\) 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 \(\text{multirember}\) is correct?

\[
\begin{align*}
\text{(define multirember} & \quad \text{(lambda} \quad (a \ \text{lat}) \\
& \quad ((Y \ (\text{lambda} \ (\text{mr}) \\
& \quad \quad (\text{lambda} \ (\text{lat}) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad ((\text{null?} \ \text{lat}) \ (\text{quote} ())) \\
& \quad \quad \quad \quad ((\text{eq?} \ a \ (\text{car} \ \text{lat})) \\
& \quad \quad \quad \quad (\text{mr} \ (\text{cdr} \ \text{lat}))) \\
& \quad \quad \quad \quad (\text{else} \ (\text{cons} \ (\text{car} \ \text{lat}) \\
& \quad \quad \quad \quad \quad (\text{mr} \ (\text{cdr} \ \text{lat})))))))) \\
& \quad \text{lat}))))
\end{align*}
\]

Whew, the \(Y\) combinator in the middle looks difficult.

What is this function?

\[
\begin{align*}
\text{(define ??} & \quad ((\text{lambda} \ (\text{le}) \\
& \quad ((\text{lambda} \ (f) \ (f \ f)) \\
& \quad (\text{lambda} \ (f) \\
& \quad \quad (\text{le} \ (\text{lambda} \ (x) \ ((f \ f) \ x)))))) \\
& \quad (\text{lambda} \ (\text{length}) \\
& \quad \quad (\text{lambda} \ (l) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad ((\text{null?} \ l) \ 0) \\
& \quad \quad \quad \quad (\text{else} \\
& \quad \quad \quad \quad \quad (\text{add1} \ (\text{length} \ (\text{cdr} \ l))))))))))
\end{align*}
\]

It is the function \(\text{length}\) in the style of chapter 9, using \(Y\).

\[
\begin{align*}
\text{(define length} & \quad (Y \ (\text{lambda} \ (\text{length}) \\
& \quad \quad (\text{lambda} \ (l) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad ((\text{null?} \ l) \ 0) \\
& \quad \quad \quad \quad (\text{else} \\
& \quad \quad \quad \quad \quad (\text{add1} \ (\text{length} \ (\text{cdr} \ l))))))))))
\end{align*}
\]
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.

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

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 \( \text{lat} \), we can use \( \text{a-lat} \) in the definition of \text{multirember} as long as we also re-name all occurrences of \( \text{lat} \) in the body of the \text{lambda} \ldots.

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

\[
\text{(define \text{multirember}}
\text{(lambda (a a-lat)}
\text{(mr a-lat)\}))
\]

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

\[
\text{(define \text{id}}
\text{(lambda (a)}
\text{a))}
\]

is the same as

\[
\text{(define \text{id}}
\text{(lambda (b)}
\text{b))}
\]

Yet if we used \( b \) in the definition of \text{multirember}

\[
\text{(define \text{multirember}}
\text{(lambda (b a-lat)}
\text{(mr a-lat)\}))
\]

the \( a \) in \( \text{mr} \) would no longer make any sense.

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

Okay, that explains things.

And that is precisely why we need \text{(letrec \ldots)}

What do you think is the purpose of the nested box?

It separates the two parts of \( \text{a (letrec \ldots)} \):
the naming part, which is the nested box, and the value part, which is \( \text{mr} \).

Is the nested box important otherwise?

No, the nested box is merely an annotation that we use to help distinguish the two parts of \text{(letrec \ldots)}. Once we get accustomed to the shape of \text{(letrec \ldots)}, we will stop drawing the inner box.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What do we use the naming part for?</td>
<td>The naming part defines a recursive function though unlike <strong>defined</strong> functions; a function defined in the naming part of <em>(letrec ...)</em> knows all the arguments of all the surrounding <em>(lambda ...)</em> expressions.</td>
</tr>
<tr>
<td>And the value part?</td>
<td>It tells us what the result of the <em>(letrec ...)</em> is. It may refer to the named recursive function.</td>
</tr>
<tr>
<td>Does this mean that <em>(letrec ((mr ...)) mr)</em> defines and returns a recursive function?</td>
<td>Precisely. Isn’t that a lot of parentheses for saying just that?</td>
</tr>
<tr>
<td>Yes, but they are important.</td>
<td>Okay, let’s go on.</td>
</tr>
<tr>
<td>What is the value of <em>(letrec ((mr ...)) mr) lat)</em></td>
<td>It is the result of applying the recursive function <em>mr</em> to <em>lat</em>.</td>
</tr>
<tr>
<td>What is the value of <em>(multirember a lat)</em></td>
<td><em>(apple custard linzer torte)</em>, but we already knew this.</td>
</tr>
<tr>
<td>How can we determine this value?</td>
<td>That’s more interesting.</td>
</tr>
<tr>
<td>The first line in the definition of <em>multirember</em> is no longer <em>(cond ...)</em> but <em>(letrec ((mr ...)) mr) lat)</em></td>
<td>We said that it defines the recursive function <em>mr</em> and applies it to <em>lat</em>.</td>
</tr>
<tr>
<td>What does this mean?</td>
<td>It is something we are quite familiar with: <em>(cond ...)</em>. We now ask questions the way we did in chapter 2.</td>
</tr>
<tr>
<td>What is the first line in <em>mr</em></td>
<td></td>
</tr>
</tbody>
</table>

20 Chapter 12
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 mrpie and had used it on lat

Correct, and the good thing is that no other function can refer to mrpie.

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

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 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:

\[
\begin{align*}
\text{(define rember-f} \\
\text{ (lambda (test?)} \\
\text{ (lambda (a l)} \\
\text{ (cond} \\
\text{ ((null? l) (quote ()))} \\
\text{ ((test? (car l) a) (cdr l))} \\
\text{ (else (cons (car l) \\
\text{ ((rember-f test?) a \\
\text{ (cdr l)))))))))}
\end{align*}
\]

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

\[
\begin{align*}
\text{(define rember-eq? (rember-f test?)}
\end{align*}
\]

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:

\[
\begin{align*}
\text{(define multirember-f} \\
\text{ (lambda (test?)} \\
\text{ (lambda (a lat)} \\
\text{ (cond} \\
\text{ ((null? lat) (quote ()))} \\
\text{ ((test? (car lat) a) \\
\text{ (multirember-f test?) a \\
\text{ (cdr lat))})} \\
\text{ (else (cons (car lat) \\
\text{ ((multirember-f test?) a \\
\text{ (cdr lat)))))))))}
\end{align*}
\]
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?)`

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

(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))))

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

True enough.
What is the value of \((\text{multirember-f } test?)\) where \(test?\) is \(eq?\)?

It is the function \(\text{multirember}\):

\[
\begin{align*}
\text{multirember} & \quad \text{(define \text{multirember}} \\
& \quad \text{(letrec}} \\
& \quad \quad \text{(mr (lambda (a lat))} \\
& \quad \quad \quad \text{(cond}} \\
& \quad \quad \quad \quad \text{(null? lat) (quote ()}} \\
& \quad \quad \quad \quad \text{(eq? (car lat) a}} \\
& \quad \quad \quad \quad \text{(mr a (cdr lat))}} \\
& \quad \quad \quad \text{(else}} \\
& \quad \quad \quad \quad \text{(cons (car lat) \}
& \quad \quad \quad \quad \quad \quad \text{(mr a (cdr lat)))})}} \\
& \quad \quad \text{mr}}
\end{align*}
\]

Did you notice that no \((\text{lambda ...})\) surrounds the \((\text{letrec ...})\)?

It looks odd, but it is correct!

Could we have used another name for the function named in \((\text{letrec ...})\)?

Yes, \(mr\) is \(\text{multirember}\).

Is this another way of writing the definition? 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))))))
```

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 \texttt{(letrec ...)} with this function:

\begin{verbatim}
(define member?
  (lambda (a lat)
    ((letrec
        ((yes? (lambda (/)
            (cond
              ((null? l) #f)
              ((eq? (car l) a) #t)
              (else (yes? (cdr l))))))
      yes?)
      lat)))

Do you also like this version?
\end{verbatim}

Here is an alternative:

\begin{verbatim}
(define member?
  (lambda (a lat)
    (letrec
      ((yes? (lambda (/)
          (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 \texttt{(letrec ...)}

Yes. We are now used to the shape of \texttt{(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 \texttt{(union set1 set2)} is where
- set1 is (tomatoes and macaroni casserole)
- set2 is (macaroni and cheese)

\begin{verbatim}
(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))))))

Write \texttt{union}
\end{verbatim}
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
      ((A (lambda (set)
           (cond
            ((null? set) set2)
            ((member? (car set) set2) (U (cdr set)))
            (else (cons (car set) (U (cdr set)))))
             (else (cons (car set) (A (cdr set)))))
             (U set1)))))
```

Could we also have written it like this?

Yes.

```
(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)))))
             (else (cons (car set) (A (cdr set)))))
             (A set1)))))
```

Correct: A is just a name like U

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

Does it matter what name we use?

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.
<table>
<thead>
<tr>
<th>Can you think of a better name for $U$</th>
<th>This should be an old shoe by now.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Now, does it work?</td>
<td>It should.</td>
</tr>
<tr>
<td>Explain in your words how the new version of union works.</td>
<td>Our words: “First, we define another function $U$ that cdrs down set, consing 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.”</td>
</tr>
<tr>
<td>How does $U$ know about set2</td>
<td>Since $U$ is defined using (letrec ...) inside of union, it knows about all the things that union knows about.</td>
</tr>
<tr>
<td>And does it have to pass around set2</td>
<td>No, it does not.</td>
</tr>
<tr>
<td>How does $U$ know about member?</td>
<td>Everyone knows the function member?.</td>
</tr>
<tr>
<td>Does it mean that the definition of union depends on the definition of member?</td>
<td>It does, but member? works, so this is no problem.</td>
</tr>
<tr>
<td>Suppose we had defined member? as follows.</td>
<td>But this would confuse union!</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
``` Scheme
(define member?
  (lambda (lat a)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (member? (cdr lat) a)))))
```
| 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:

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

Fill in the dots.

```lisp
... ((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? Presumably.

```lisp
(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))))
```

---

*Take Cover*
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?

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

```
(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))))
```

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:

```scheme
(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 (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-b* 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?

Yes, but can’t this wait?

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

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

How about black currant sorbet?
13.
Hop, Skip, and Jump
What is the value of \((\text{intersect } \text{set1 } \text{set2})\)
where
\(\text{set1}\) is (tomatoes and macaroni)
and
\(\text{set2}\) is (macaroni and cheese)

Is \text{intersect} an old acquaintance?

Yes, we have known \text{intersect} for as long as we have known \text{union}.

Write \text{intersect}

Sure, here we go:

\[
(\text{define intersect}
(\lambda (\text{set1} \text{set2})
(\text{cond}
  ((\text{null? set1}) (\text{quote ()}))
  ((\text{member? (car set1) set2})
   (\text{cons (car set1)}
    (\text{intersect (cdr set1) set2})))
  (\text{else (intersect (cdr set1) set2))])))
\]

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

\[
(\text{define intersect}
(\lambda (\text{set1} \text{set2})
(\text{letrec}
  ((I (\lambda (\text{set})
      (\text{cond}
        ((\text{null? set}) (\text{quote ()}))
        ((\text{member? (car set) set2})
         (\text{cons (car set)}
          (I (cdr set))))
        (\text{else (I (cdr set))})))
     (I set1)))))
\]
Do you also recall `intersectall`

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

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

Why don't we ask `(null? iset)`

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? iset)` and then just use the two `cond`-lines from the earlier `intersectall`:

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

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 `iset` 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 ...)

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

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

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 \((\text{intersectall } lset)\) where
\(lset\) is \(((3 \text{ mangos and}) \to (3 \text{ kiwis and}) \to (3 \text{ hamburgers}))\)

\((3)\).

What is \((\text{intersectall } lset)\) where
\(lset\) is \(((3 \text{ steaks and}) \to (\text{no food and}) \to (3 \text{ baked potatoes}) \to (3 \text{ diet hamburgers}))\)

\(()\).

What is \((\text{intersectall } lset)\) where
\(lset\) is \(((3 \text{ mangoes and}) \to () \to (3 \text{ diet hamburgers}))\)

\(()\).

Why is this?

The intersection of \((3 \text{ mangos and}), (), \text{ and } (3 \text{ diet hamburgers})\) is the empty set.

Why is this?

When there is an empty set in the list of sets, \((\text{intersectall } lset)\) returns the empty set.

But this does not show how \text{intersectall} determines that the intersection is empty.

No, it doesn't. Instead, it keeps \text{intersecting} the empty set with some set until the list of sets is exhausted.

Wouldn’t it be better if \text{intersectall} didn’t have to \text{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 \((\text{intersect } lset)\).
Well, there actually is a way to say such things.

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

Because we did not need it until now.

That’s simple. Here we go:

(define intersectall
  (lambda (lset)
    (letcc 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))))))

Alonzo Church (1903–1995) would have written:

(define intersectall
  (lambda (lset)
    (call-with-current-continuation
      (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))))))))
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?

But the first cond-line in A should be obvious: we ask one extra question

\( (\text{null? (car lset))} \)

and if it is true, A uses hop as if it were a function.

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

What is the value of \((\text{intersectall lset})\) where

\( lset \) is \(((3 \text{ mangoes and}) \)

\( () \)

\( (3 \text{ diet hamburgers}) \)

So how do we determine the answer for \((\text{intersectall lset})\) \n
Since we don’t know what this line does, it is probably best to ignore it for the time being. What next?

And so we go on and ... ... determine the value of \((A lset)\) where \(lset\) is the list of sets.

What is the next question?

Is this true?
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is this why we ask <code>(null? (cdr lset))</code></td>
<td>Yes, and it is not true either.</td>
</tr>
<tr>
<td><strong>else</strong></td>
<td>Of course.</td>
</tr>
<tr>
<td>And now we recur?</td>
<td>Yes, we remember that <code>(car lset)</code> is (3 mangos and), and that we must <strong>intersect</strong> this set with the result of <code>(A (cdr lset))</code>.</td>
</tr>
</tbody>
</table>
| 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:                                                       | But how do we forget something?                                        |
| “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.” |                                                                         |
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 intersecting 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

\( lset = ((3\text{ steaks and})
\quad (no\text{ food and})
\quad (three\text{ baked potatoes})
\quad (3\text{ diet hamburgers})) \)

We ignore (letcc hop).

And then?

We determine the value of (A lset) because lset is not empty.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What do we ask next?</td>
<td><em>(null? (car lset)), which is false.</em></td>
</tr>
<tr>
<td>And next?</td>
<td><em>(null? (cdr lset)), which is false.</em></td>
</tr>
</tbody>
</table>
| 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 potatoes) 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.
<table>
<thead>
<tr>
<th>Oh, yes. We must now determine the value of (letcc hop (quote ()))</th>
<th>That’s correct. But what does this line do now that we did not use hop?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nothing.</td>
<td>What do you mean, nothing?</td>
</tr>
<tr>
<td>When we need to determine the value of (letcc hop (quote ())) there is nothing left to do. We know the value.</td>
<td>You mean, it is () again?</td>
</tr>
<tr>
<td>Yes, it is () again.</td>
<td>That’s simple.</td>
</tr>
<tr>
<td>Isn’t it?</td>
<td>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.</td>
</tr>
<tr>
<td>Is it a mistake of intersectall</td>
<td>Yes, and it is also a mistake of intersect.</td>
</tr>
<tr>
<td>In what sense?</td>
<td>We could have defined intersect so that it would not do anything when its second argument is the empty set.</td>
</tr>
<tr>
<td>Why its second argument?</td>
<td>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!</td>
</tr>
</tbody>
</table>
Should we have defined `intersect` with an extra question about `set2`?

```
(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)))))))
      (cond
        ((null? set2) (quote ()))
        (else (I set1))))))
```

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

Yes.

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

What is the value of `(intersectall Iset)`

We know it is `()`. Where

`Iset` is `((3 steaks and) (no food and) (three baked potatoes) (3 diet hamburgers))`

Should we go through the whole thing again?

We could skip the part when `A` looks at all the sets until `Iset` 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 Iset)` 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 Is to go back and pick up. |
| We need to determine the value of |
| a. I of (three baked potatoes) 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 \textit{rember} with \((letrec \ldots)\)  

Sure can:

\begin{verbatim}
(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))))
\end{verbatim}

52  

Chapter 13
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)

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 \((\text{rember-upto-last a lat})\) where \(a\) is roots and \(\text{lat}\) is \((\text{noodles spaghetti spätzle bean-thread roots potatoes yam others rice})\)?

And \((\text{rember-upto-last a lat})\) where \(a\) is sweetthing and \(\text{lat}\) is \((\text{noodles spaghetti spätzle bean-thread roots potatoes yam others rice})\).

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

Can you describe in two sentences what \(\text{rember-upto-last}\) does?

Here are our two sentences:

"The function \(\text{rember-upto-last}\) 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 \(a\), \(\text{rember-upto-last}\) returns the lat."
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does this sound like yet another version of <code>rember</code>?</td>
<td>Yes, it does.</td>
</tr>
<tr>
<td>How would you change the function $R$ in <code>rember</code> or <code>rember-beyond-first</code> to get <code>rember-upto-last</code>?</td>
<td>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.</td>
</tr>
<tr>
<td>You mean it would forget some computation that it had remembered somewhere?</td>
<td>Yes, it would.</td>
</tr>
<tr>
<td>Does this sound like <code>intersectall</code>?</td>
<td>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.</td>
</tr>
<tr>
<td>Different in what sense?</td>
<td>The function <code>intersectall</code> knows what the result is; <code>rember-upto-last</code> knows which pieces of the list are <em>not</em> in the result.</td>
</tr>
<tr>
<td>But does it know where it can find the result?</td>
<td>The result is the <code>rember-upto-last</code> of the rest of the list.</td>
</tr>
<tr>
<td>Suppose <code>rember-upto-last</code> sees the atom $a$ should it forget the pending computations, and should it restart the process of searching through the rest of the list?</td>
<td>Yes, it should.</td>
</tr>
<tr>
<td>We can do this.</td>
<td>You mean we could use (letcc ...) to do this, too?</td>
</tr>
<tr>
<td>Yes.</td>
<td>How would it continue searching, but ignore the atoms that are waiting to be consed onto the result?</td>
</tr>
</tbody>
</table>
How would you say, “Do this or that to the rest of the list”?

Easy: do this or that to \((cdr\ \text{lat})\).

And how would you say “Ignore something”? With a line like \((\text{skip} \ldots)\), assuming the beginning of the function looks like \((\text{letcc} \ \text{skip})\).

Well then … … if we had a line like 
\[(\text{letcc} \ \text{skip})\]
at the beginning of the function, we could say 
\[(\text{skip} \ (R \ (cdr\ \text{lat})))\]
when necessary.

Yes, again. Can you write the function \(\text{rember-upto-last}\) now? Yes, this must be it:

\[
\begin{align*}
(\text{define} & \ \text{rember-upto-last} \\
& (\lambda (a\ \text{lat}) \\
& (\text{letcc} \ \text{skip} \\
& (\text{letrec} \\
& ((R\ (\lambda (\text{lat}) \\
& \quad (\text{cond} \\
& \quad\quad ((\text{null?} \ \text{lat}) \ (\text{quote} \ ())) \\
& \quad\quad ((\text{eq?} \ (\text{car} \ \text{lat}) \ a) \\
& \quad\quad \ (\text{skip} \ (R\ (cdr\ \text{lat})))) \\
& \quad\quad \text{else} \\
& \quad\quad \ (\text{cons} \ (\text{car} \ \text{lat}) \\
& \quad\quad \quad (R\ (cdr\ \text{lat})))))))))) \\
& \quad\quad (R\ \text{lat}))))))
\end{align*}
\]

Ready for an example? Yes, let’s try the one with the sweet things.
You mean the one where a is cookies and

\[ \text{lat is (cookies}
\]

\[ \hspace{1cm} \text{chocolate mints}
\]

\[ \hspace{1cm} \text{caramel delight ginger snaps}
\]

\[ \hspace{1cm} \text{desserts}
\]

\[ \hspace{1cm} \text{chocolate mousse}
\]

\[ \hspace{1cm} \text{vanilla ice cream}
\]

\[ \hspace{1cm} \text{German chocolate cake}
\]

\[ \hspace{1cm} \text{more cookies}
\]

\[ \hspace{1cm} \text{gingerbreadman chocolate}
\]

\[ \hspace{1cm} \text{chip brownies)}\]

Yes, that’s the one.

No problem. What is the first thing we do?  
We see \((\text{letcc skip})\) and ignore it for a while.

Great. And then?  
We ask \((\text{null? lat})\).

Why?  
Because we use \(R\) to determine the value of \((\text{rember-upto-last a lat})\).

And \((\text{null? lat})\) is not true.  
But \((\text{eq? (car lat) a})\) is true.

Which means we \text{skip} and actually determine the value of

\[(\text{letcc skip (R (cdr lat))})\]

where

\[ \text{lat is (cookies}
\]

\[ \hspace{1cm} \text{chocolate mints}
\]

\[ \hspace{1cm} \text{caramel delight ginger snaps}
\]

\[ \hspace{1cm} \text{desserts}
\]

\[ \hspace{1cm} \text{chocolate mousse}
\]

\[ \hspace{1cm} \text{vanilla ice cream}
\]

\[ \hspace{1cm} \text{German chocolate cake}
\]

\[ \hspace{1cm} \text{more cookies}
\]

\[ \hspace{1cm} \text{gingerbreadman chocolate}
\]

\[ \hspace{1cm} \text{chip brownies)}\]

Yes.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What next?</td>
<td>We ask ((null? \text{lat})).</td>
</tr>
<tr>
<td>Which is not true.</td>
<td>And neither is ((eq? (\text{car lat}) a)).</td>
</tr>
<tr>
<td>So what?</td>
<td>We recur.</td>
</tr>
<tr>
<td>How?</td>
<td>We remember to \text{cons} \text{chocolate} onto the result of ((R (\text{cdr lat}))) where \text{lat} is (chocolate mints caramel delight ginger snaps desserts chocolate mousse vanilla ice cream German chocolate cake more cookies gingerbreadman chocolate chip brownies).</td>
</tr>
<tr>
<td>Next?</td>
<td>Well, this goes on for a while.</td>
</tr>
<tr>
<td>You mean it drags on and on with this recursion.</td>
<td>Exactly.</td>
</tr>
<tr>
<td>Should we gloss over the next steps?</td>
<td>Yes, they're pretty easy.</td>
</tr>
<tr>
<td>What should we look at next?</td>
<td>We should remember to \text{cons} \text{chocolate}, mints, caramel, delight, ginger, snaps, desserts, chocolate, mousse, vanilla, ice, cream, German, chocolate, cake, and more onto the result of ((R (\text{cdr lat}))) where \text{lat} is (more cookies gingerbreadman chocolate chip brownies). And we must not forget the (\text{letcc skip} ... at the end!)</td>
</tr>
</tbody>
</table>

*Hop, Skip, and Jump* 59
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
conses 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 \textit{leftmost} \hfill Is it the function that extracts the leftmost atom from a list of S-expressions?

Yes, and here is the definition: \hfill Okay.
\begin{verbatim}
(define leftmost
  (lambda (l)
      (cond
        ((atom? (car l)) (car l))
        (else (leftmost (car l)))))))
\end{verbatim}

What is the value of \((\text{leftmost } l)\) where \(l\) is \(((a) b) (c d))\)? \hfill a, of course.

And what is the value of \((\text{leftmost } l)\) where \(l\) is \(((a) ()) () (e))\)? \hfill It's still a.

How about this: \((\text{leftmost } l)\) where \(l\) is \(((()) a) ())\) \hfill It should still be a, but there is actually no answer.

Why is it not a \hfill In chapter 5, we said that the function \textit{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 \((\text{leftmost } l)\) where the list \(l\) contained an empty list? \hfill Yes, we did: \(l\) was \(((a) ()) () (e))\).

Shouldn't we be able to define a version of \textit{leftmost} that does not restrict the shape of its argument? \hfill 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?

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

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

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 ().
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value of ((\text{leftmost} \ (\text{quote} \ ())))</td>
<td>It is (), which means that we haven't found an atom yet.</td>
</tr>
<tr>
<td>What do we need to do?</td>
<td>We also need to recur with the (\text{cdr}) of the list, if we can't find an atom in the (\text{car}).</td>
</tr>
<tr>
<td>How do we determine whether ((\text{leftmost} \ (\text{car} \ l))) found an atom?</td>
<td>We ask ((\text{atom?} \ (\text{leftmost} \ (\text{car} \ l))))), because \text{leftmost} only returns an atom if its argument contains one.</td>
</tr>
<tr>
<td>And when ((\text{atom?} \ (\text{leftmost} \ (\text{car} \ l))))) is true?</td>
<td>Then we know what the leftmost atom is.</td>
</tr>
<tr>
<td>And how do we say it?</td>
<td>Easy: ((\text{leftmost} \ (\text{car} \ l)))).</td>
</tr>
<tr>
<td>But if ((\text{atom?} \ (\text{leftmost} \ (\text{car} \ l))))) is false?</td>
<td>Then we continue to look for an atom in the (\text{cdr}) of (l).</td>
</tr>
</tbody>
</table>
| Define \text{leftmost}                                                | \[
\begin{align*}
\text{define leftmost} \\
& \quad \text{(lambda} \ (l) \\
& \quad \quad \text{(cond} \\
& \quad \quad \quad ((\text{null?} \ l) \ (\text{quote} \ ())) \\
& \quad \quad \quad ((\text{atom?} \ (\text{car} \ l)) \ (\text{car} \ l)) \\
& \quad \quad \quad \text{(else} \ (\text{cond} \\
& \quad \quad \quad \quad ((\text{atom?} \ (\text{leftmost} \ (\text{car} \ l)))) \\
& \quad \quad \quad \quad \quad \ (\text{leftmost} \ (\text{car} \ l))) \\
& \quad \quad \quad \quad \quad \quad \ (\text{else} \ (\text{leftmost} \ (\text{cdr} \ l))))))))
\end{align*}
\]
| \((\text{leftmost} \ l)\) where \(l\) is \(((a) \ b) \ (c \ d))\) | a.                                                                     |
| \((\text{leftmost} \ l)\) where \(l\) is \(((a) () () (e))\)          | a.                                                                     |
(leftmost l)
where
l is (((() a) () ()))

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:

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 ((a (leftmost (car l))))
   (cond
     ((atom? a) a)
     (else (leftmost (cdr l))))))
...
```

1 Like (and ... ), (let ... ) is an abbreviation:

```
(let ((x_1 \alpha_1) ... (x_n \alpha_n)) \beta ... )
= ((\lambda (x_1 ... x_n) \beta ... ) \alpha_1 ... \alpha_n)
```
Isn't this much easier to read? Yes, it is.

What is the value of \( \text{rember1* } a \ l \) where \( a \) is salad and
\( l \) is \((\text{Swedish rye}) \)
\( (\text{French (mustard salad turkey)}) \)
\( \text{salad} \).

\((\text{rember1* } a \ l)\) where \( a \) is meat and
\( l \) is \((\text{pasta meat}) \)
\( \text{pasta} \)
\( (\text{noodles meat sauce}) \)
\( \text{meat tomatoes} \).

Take a close look at \( \text{rember1*} \)

\[
\text{(define rember1*}
\begin{align*}
\text{(lambda } & (a \ l) \text{)} \\
\text{cond} & \text{ (null? } l \text{) (quote }) \\
\text{cond} & (\text{atom? (car } l\text{)}) \\
\text{cond} & ((\text{eq? (car } l \text{) a} ) \ (\text{cdr } l)) \\
\text{else} & \text{(cons (car } l \text{) (rember1* } a \ \text{(cdr } l)))))) \\
\text{else} & \text{ (cond} \\
\text{eqlist?} & \text{(rember1* } a \ \text{(car } l\text{))} \\
\text{cond} & \text{(car } l\text{)} \\
\text{else} & \text{(cons (car } l \text{) (rember1* } a \ \text{(cdr } l)))))) \\
\text{else} & \text{(cons (rember1* } a \ \text{(car } l\text{))} \\
\text{cond} & \text{(cdr } l)))))) \\
\text{Fix rember1* using The Twelfth Commandment.}
\end{align*}
\]

It even has the same expressions underlined.

\[
\text{(define rember1*}
\begin{align*}
\text{(letrec (R (lambda } & (l) \text{)} \\
\text{cond} & \text{(null? } l \text{) (quote }) \\
\text{cond} & (\text{atom? (car } l\text{)}) \\
\text{cond} & ((\text{eq? (car } l \text{) a} ) \ (\text{cdr } l)) \\
\text{else} & \text{(cons (car } l \text{) (R (cdr } l)))))) \\
\text{else} & \text{ (cond} \\
\text{eqlist?} & \text{(R (car } l\text{))} \\
\text{cond} & \text{(car } l\text{)} \\
\text{else} & \text{(cons (car } l \text{) (R (cdr } l))))) \\
\text{else} & \text{(cons (R (car } l\text{))} \\
\text{cond} & \text{(cdr } l)))))) \\
\text{(R } l))) \\
\end{align*}
\]

Let There Be Names 67
**What does** \((\text{rember1* } a \ l)\)** 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 \((R (\text{car } l))\) with \((\text{car } l)\)?**

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 \((\text{car } l)\) when \((\text{car } l)\) is a list. If it doesn’t find one, it continues its search; otherwise, that atom is the result.

**Do the underlined instances of **\((R (\text{car } l))\)** 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 **\((\text{let} \ldots)\)**

\[
\begin{align*}
(\text{define } \text{rember1*} \\
(\lambda (a \ l) \\
(\text{letrec} \\
(R (\lambda (l) \\
(\text{cond} \\
((\text{null? } l) (\text{quote } ())) \\
((\text{atom? } (\text{car } l)) \\
(\text{cond} \\
(((\text{eq? } (\text{car } l) a) (\text{cdr } l)) \\
(\text{else } (\text{cons } (\text{car } l) \\
(R (\text{cdr } l)))))))) \\
(\text{else } \ldots)))))))) \\
(R l))))
\end{align*}
\]

Here is the rest of the minor function **R**

\[
\begin{align*}
(\text{let } ((\text{av } (R (\text{car } l)))) \\
(\text{cond} \\
(((\text{eqlist? } (\text{car } l) \text{ av}) \\
(\text{cons } (\text{car } l) (R (\text{cdr } l)))))) \\
(\text{else } (\text{cons } \text{ av } (\text{cdr } l))))))
\end{align*}
\]

...
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 letting. 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.

It looks like a normal *-function.

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

Let’s try an example. Determine the value of 2.

(depth* l)

where

l is ((pickled) peppers (peppers pickled))

Let There Be Names
Here is another one: \((depth* l)\) where
\(l\) is (margarine
  ((bitter butter)
   (makes)
   (batter (bitter)))
  butter)

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 \((\text{let} \ldots)\).

What should we use \((\text{let} \ldots)\) for?
We determine the value of \((depth* (\text{car} l))\)
and the value of \((depth* (\text{cdr} l))\) 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.

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

How about this one?

Should we try some examples?
It should be correct. Using \((\text{let} \ldots)\) is straightforward.
Let's try it anyway. What is the value of 
\[(depth* l)\]
where
\[l \text{ is } (())\]
\[((\text{bitter butter})\text{ makes})\text{ (batter (bitter))) butter}\]
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 
\[(addl (depth* (car l)))\text{ and } (depth* (cdr l))\] 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 
\[(addl (depth* (car l)))\]
where
\[l \text{ is } (())\]
\[((\text{bitter butter})\text{ makes})\text{ (batter (bitter))) butter}\).

How do we do that?
We use \(depth^\ast\) 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 
\[(addl (depth* (car l)))\text{ where } l \text{ is } ()\].

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))))))))
```

Why does this version of depth* work?

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

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 
\((\text{remember}\ l\ ((\text{car}\ l)))\) twice if we hadn’t introduced a name for its value?

Yes.

How should we use \((\text{let } \ldots)\) in \(\text{depth}\) if we want to use it right after finding out whether or not \(l\) is empty?

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

Let’s do it! Here is an outline.

\[
\begin{align*}
\text{(define depth*} & \\
& \text{(lambda} (l) \\
& \text{(cond} \\
& \quad ((\text{null}\ l) 1) \\
& \quad \text{(else } \ldots)))))
\end{align*}
\]

Fill in the dots.

\[
\begin{align*}
\text{(define depth*} & \\
& \text{(lambda} (l) \\
& \text{(cond} \\
& \quad ((\text{null}\ l) 1) \\
& \quad \text{(else } \ldots)))))
\end{align*}
\]

\[
\begin{align*}
\text{(let} & ((d (\text{depth}\ (\text{cdr}\ l)))) \\
& \text{(cond} \\
& \quad ((\text{atom}\ (\text{car}\ l)) d) \\
& \quad \text{(else} \\
& \quad \quad \text{(cond} \\
& \quad \quad \quad ((> d (\text{addl} (\text{depth}\ (\text{car}\ l)))) d) \\
& \quad \quad \quad \text{(else} (\text{addl} (\text{depth}\ (\text{car}\ l)))))) \\
& \quad \quad \quad \quad )))) \\
& \quad )))
\end{align*}
\]

And when can we use \((\text{let } \ldots)\) for the repeated expression \((\text{addl} (\text{depth}\ (\text{car}\ l)))\)

When we know that \((\text{car}\ l)\) is not an atom:

\[
\begin{align*}
\text{(define depth*} & \\
& \text{(lambda} (l) \\
& \text{(cond} \\
& \quad ((\text{null}\ l) 1) \\
& \quad \text{(else } \ldots)))))
\end{align*}
\]

\[
\begin{align*}
\text{(let} & ((d (\text{depth}\ (\text{cdr}\ l)))) \\
& \text{(cond} \\
& \quad ((\text{atom}\ (\text{car}\ l)) d) \\
& \quad \text{(else} \\
& \quad \quad \text{(let} ((a (\text{addl} (\text{depth}\ (\text{car}\ l)))))) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad ((> d a) d) \\
& \quad \quad \quad \quad \text{(else} a)))))) \\
& \quad )))
\end{align*}
\]

Would we have needed to determine \((\text{depth}\ (\text{cdr}\ l))\) twice if we hadn’t introduced a name for its value?

No. If the first element of \(l\) is an atom, \((\text{depth}\ (\text{cdr}\ l))\) is evaluated only once.

If it doesn’t help to name the value of \((\text{depth}\ (\text{cdr}\ l))\) we should check whether the new version of \(\text{depth}\) is easier to read.

Not really. The three nested \text{conds} hide what kinds of data the function sees.

Let There Be Names
So which version of $\text{depth}^*$ is our favorite version?

\[
\text{(define } \text{depth}^* \\
\quad \text{(lambda } (l) \\
\quad \quad \text{(cond} \\
\quad \quad \quad ((\text{null? } l) 1) \\
\quad \quad \quad ((\text{atom? } (\text{car } l)) \\
\quad \quad \quad \quad (\text{depth}^* (\text{cdr } l))) \\
\quad \quad \quad (\text{else} \\
\quad \quad \quad \quad \text{(let } ((a (\text{add1 } (\text{depth}^* (\text{car } l)))) \\
\quad \quad \quad \quad \quad (d (\text{depth}^* (\text{cdr } l)))) \\
\quad \quad \quad \quad \quad \quad \text{(cond} \\
\quad \quad \quad \quad \quad \quad \quad ((> d a) d) \\
\quad \quad \quad \quad \quad \quad \quad (\text{else } a)))))))
\]

---

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 $\text{depth}^*$ 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 letting things be the way they are supposed to be. Oh, yes. And we sure did.

Can we make $\text{depth}^*$ 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

```
(define (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)))))
```

Yes, no problem.

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

Let There Be Names 75
Here is another chance to practice letting: do it for the protected version of `scramble` from chapter 12:

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

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.

```scheme
(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`.

---

Chapter 14
<table>
<thead>
<tr>
<th>And how do we determine this?</th>
<th>We have done this before.</th>
</tr>
</thead>
<tbody>
<tr>
<td>So how do we do it?</td>
<td>We quickly determine that ( l ) isn't empty and doesn't contain an atom in the \textit{car} position. So we recur with (\textit{leftmost} ( l )) where ( l ) is ((a)).</td>
</tr>
<tr>
<td>What do we do next?</td>
<td>We quickly determine that ( l ) isn't empty and doesn't contain an atom in the \textit{car} position. So we recur with (\textit{leftmost} ( l )) where ( l ) is (a).</td>
</tr>
<tr>
<td>And now?</td>
<td>Now (\textit{car} ( l )) is a, so we are done.</td>
</tr>
<tr>
<td>Are we really done?</td>
<td>Well, we have the value for (\textit{leftmost} ( l )) where ( l ) is (a).</td>
</tr>
<tr>
<td>What do we do with this value?</td>
<td>We name it ( a ) and check whether it is an atom. Since it is an atom, we are done.</td>
</tr>
<tr>
<td>Are we really, really done?</td>
<td>Still not quite, but we have the value for (\textit{leftmost} ( l )) where ( l ) is ((a)).</td>
</tr>
<tr>
<td>And what do we do with this value?</td>
<td>We name it ( a ) again and check whether it is an atom. Since it is an atom, we are done.</td>
</tr>
<tr>
<td>So, are we done now?</td>
<td>No. We need to name ( a ) one more time, check that it is an atom one more time, and then we're completely done.</td>
</tr>
</tbody>
</table>

\textit{Let There Be Names}
Have we been here before?

Yes, we have. When we discussed \textit{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 \texttt{(letcc ...)}.

Here is a new definition of \textit{leftmost}

\begin{verbatim}
(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 ()
              (lm (car l) out)
              (lm (cdr l) out))))))
\end{verbatim}

Did you notice the unusual \texttt{(let ...)}

Yes, the \texttt{(let ...)} contains two expressions in the value part.

What are they?

The first one is
\texttt{(lm (car l) out)}.
The one after that is
\texttt{(lm (cdr l) out)}.

\footnote{L: \texttt{progn} also works.\newline
S: \texttt{begin} also works.}
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 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 \textit{car} position. So we recur with \((lm \ l \ out)\)

where

\( l \) is (a)

and

\( out \) is \textit{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 \textit{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 \textit{skip}, the needle of a compass as well as \((lm \ l \ out)\)

where

\( l \) is (b (c))

and

\( out \) is \textit{skip}, the needle of a compass.

Don’t we have an atom in \textit{car} of \( l \) now?

We do. And that means we need to understand

\((out \ (car \ l))\)

where

\( l \) is (a)

and

\( out \) is \textit{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

\( \textit{letcc} \ \textit{skip} \ a \)

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 \textit{leftmost}
Yes, that’s true. We shouldn’t forget The Thirteenth Commandment when we use The Fourteenth.

Here is one way to hide \(lm\)

\[
\begin{align*}
\text{(define leftmost}
\text{(letrec}
\text{((lm (lambda (l out)
\text{  (cond}
\text{  ((null? l) (quote ()))
\text{  ((atom? (car l))
\text{    (out (car l)))
\text{  (else}
\text{    (let ()
\text{      (lm (car l) out))
\text{    (lm (cdr l) out))))))})
\text{(lambda (l))
\text{  (letcc skip
\text{    (lm l skip))))})})})
\end{align*}
\]

Can you think of another?

In chapter 12 we usually moved the minor function out of a (\textit{lambda} ...)’s value part, but we can also move it in:

\[
\begin{align*}
\text{(define leftmost}
\text{(lambda (l))
\text{(letrec}
\text{((lm (lambda (l out)
\text{  (cond}
\text{  ((null? l) (quote ()))
\text{  ((atom? (car l))
\text{    (out (car l)))
\text{  (else}
\text{    (let ()
\text{      (lm (car l) out))
\text{    (lm (cdr l) out))))))})
\text{(letcc skip
\text{    (lm l skip))))})})
\end{align*}
\]

Correct! Better yet: we can move the (\textit{letrec} ...) into the value part of the (\textit{letcc} ...)

\[
\begin{align*}
\text{(define leftmost}
\text{(lambda (l))
\text{ (letcc skip
\text{  (letrec (...
\text{  (lm l skip))))})})
\end{align*}
\]

Can you complete the definition?

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 \( skip \)

Yes, all names are equal.

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 \((letrec \ldots)\) so that everything works:

\[
\begin{align*}
\text{(define leftmost} & \\
& \text{(lambda} \ (l) \\
& \text{(letcc} \ skip \\
& \text{(letrec} \ldots) \\
& \text{(lm} \ l \ skip))))))
\end{align*}
\]

\[
\begin{align*}
\text{(define leftmost} & \\
& \text{(lambda} \ (l) \\
& \text{(letcc} \ skip \\
& \text{letrec} \\
& \text{(lm} \ (\text{lambda} \ (l) \\
& \text{(cond} \\
& \text{((null?} \ l) \ (quote )))) \\
& \text{((atom?} \ (\text{car} \ l)) \\
& \text{(skip} \ (\text{car} \ l)))) \\
& \text{else} \\
& \text{(let} () \\
& \text{(lm} \ (\text{car} \ l) \ skip \\
& \text{(lm} \ (\text{cdr} \ l) \ skip)))))))))
\end{align*}
\]
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.)

<table>
<thead>
<tr>
<th>Didn’t we say that <code>leftmost</code> and <code>rember1*</code> are related?</th>
<th>Yes, we did.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is <code>rember1*</code> also a function that finds the final result yet checks many times that it did?</td>
<td>No, in that regard <code>rember1*</code> is quite different. Every time it finds that the <code>car</code> of a list is a list, it works through the <code>car</code> and checks right afterwards with <code>eqlist?</code> whether anything changed.</td>
</tr>
<tr>
<td>Does <code>rember1*</code> know when it failed to accomplish anything?</td>
<td>It does: every time it encounters the empty list, it failed to find the atom that is supposed to be removed.</td>
</tr>
<tr>
<td>Can we help <code>rember1*</code> by using a compass needle when it finds the empty list?</td>
<td>With the help of a North Pole and a compass needle, we could abruptly and promptly signal that the list in the <code>car</code> of a list did not contain the interesting atom.</td>
</tr>
</tbody>
</table>
Here is a sketch of the function \( rm \) which takes advantage of this idea:

\[
\begin{align*}
\text{(define } rm & \text{)} \\
& \quad \text{(lambda } (a \ l \ oh) \text{)} \\
& \quad \text{(cond)} \\
& \quad \quad ((\text{null? } l) \ (oh \ (quote \ no))) \\
& \quad \quad ((\text{atom? } (car l)) \ (cons \ (car l) \ (rm \ a \ (cdr l) \ oh)))) \\
& \quad \quad (\text{else} \ldots \\
& \quad \quad \quad \text{(letcc } oh \text{)} \\
& \quad \quad \quad \quad \quad (\text{rm } a \ (\text{car } l) \ \text{oh})) \\
& \quad \quad \quad \ldots)))))
\end{align*}
\]

What does the function do when it encounters a list in \( (\text{car } l) \)?

What kind of value does \( \text{(letcc } oh \text{)} \ (\text{rm } a \ (\text{car } l) \ \text{oh}) \) yield when \( (\text{car } l) \) does not contain \( a \)?

And what kind of value do we get when the \( \text{car} \) of \( l \) contains \( a \)?

Then what do we need to check next?

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

\[
\begin{align*}
\text{(atom?)} \\
& \quad \text{(letcc } oh \text{)} \\
& \quad \quad (\text{rm } a \ (\text{car } l) \ \text{oh}))
\end{align*}
\]

And then?

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

How do we try to remove the leftmost occurrence of \( a \) in \( (\text{cdr } l) \)?

Easy: with \( (\text{rm } a \ (\text{cdr } l) \ \text{oh}) \).
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
</table>
| Is this the only thing we have to do?                                   | No, we must not forget to add on the unaltered \((\text{car } l)\) when we succeed. We can do this with a simple \(\text{cons}\):  
\[
(\text{cons} \ (\text{car } l) \ (\text{rm } a \ (\text{cdr } l) \ oh)).
\]  |
| And if \((\text{letcc } oh \ldots)\)'s value is not an atom?            | Then it is a list, which means that \(\text{rm}\) succeeded in removing the first occurrence of \(a\) from \((\text{car } l)\). |
| How do we build the result in this case?                               | We \(\text{cons}\) the very value that \(\text{letcc } oh\)  
\[
(\text{letcc } oh
(\text{rm } a \ (\text{car } l) \ oh))
\]  produced onto \((\text{cdr } l)\), which does not change. |
| Which compass needle do we use to reconstruct this value?               | We don't need one because we know \(\text{rm}\) will succeed in removing an atom.                                                    |
| Does this mean we can use \((\text{rm } a \ (\text{car } l) \ 0)\)       | Yes, any value will do, and \(0\) is a simple argument.                                                                               |
| Let's do that!                                                         | Here is a better version of \(\text{rm}\):                                                                                           |

\[
(\text{define } \text{rm}
(\lambda (a l oh)
(\text{cond}
  ((\text{null? } l) (oh (quote no)))
  ((\text{atom?} (\text{car} l))
   (if (eq? (\text{car} l) a)
    (\text{cdr} l)
    (\text{cons} (\text{car} l)
     (\text{rm} a (\text{cdr} l) oh))))))
  (\text{else}
   (if (\text{atom}?)
    (\text{letcc} oh
     (\text{rm} a (\text{car} l) oh))
    (\text{cons} (\text{rm} a (\text{car} l))
     (\text{rm} a (\text{cdr} l) oh))
    (\text{cons} (\text{rm} a (\text{car} l) 0)
     (\text{cdr} l))))))
)\]

---

\(\text{Let There Be Names}\) 85
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How can we use \texttt{rm}</td>
<td>We need to set up a North Pole first.</td>
</tr>
<tr>
<td>Why?</td>
<td>If the list does not contain the atom we want to remove, we must be able to say no.</td>
</tr>
<tr>
<td>What is the value of ((\texttt{letcc \texttt{Say (rm a l Say)})})</td>
<td>(((\text{food}) \text{ more (food))})</td>
</tr>
<tr>
<td>where (a) is noodles and (l) is (((\text{food}) \text{ more (food))})</td>
<td>because this list does not contain noodles.</td>
</tr>
<tr>
<td>And how do we determine this?</td>
<td>Since ((\text{car } l)) is a list, we set up a new North Pole, called \textit{oh}, and recur with ((\texttt{rm a (car l) oh})) where (a) is noodles and (l) is (((\text{food}) \text{ more (food))}).</td>
</tr>
<tr>
<td>Which means?</td>
<td>After one more recursion, using the second \texttt{cond}-line, \texttt{rm} is used with noodles, the empty list, and the compass needle \textit{oh}. Then it forgets the pending \texttt{cons} of food onto the result of the recursion and checks whether \texttt{no} is an atom.</td>
</tr>
<tr>
<td>And \texttt{no} is an atom ...</td>
<td>Yes, it is. So we recur with ((\texttt{cons (car l) (rm a (cdr l) Say)})) where (a) is noodles and (l) is (((\text{food}) \text{ more (food))}).</td>
</tr>
<tr>
<td>How do we determine the value of ((\texttt{rm a l Say}))</td>
<td>We recur with the list (((\text{food}))) and, if we get a result, we \texttt{cons} more onto it.</td>
</tr>
<tr>
<td>where (a) is noodles and (l) is ((\text{more (food))})</td>
<td></td>
</tr>
</tbody>
</table>
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 \textit{reber1} would return the unaltered list, wouldn't it?

No problem:

\begin{verbatim}
(define reber1*
  (lambda (a l)
    (if (atom? (letcc oh (rm a l oh)))
        l
        (rm a l (quote ()))))
\end{verbatim}

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

Since \textit{rm} will succeed, any value will do, and
() is another simple argument.
Didn’t we forget to name the values of some expression in \textit{rember1*}

\begin{verbatim}
(define rember1*
  (lambda (a l)
    (let ((new-l (letcc oh (rm a l oh))))
      (if (atom? new-l)
          l
          new-l))))
\end{verbatim}

We can also use (let ...) in \textit{rm}:

\begin{verbatim}
(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))))))))
\end{verbatim}

Do we need to make up a good example for \textit{rember1*}?

\begin{tabular}{p{.5\textwidth}p{.5\textwidth}}
\textbf{We should, but aren’t we late for dinner?} & \textbf{We should, but aren’t we late for dinner?} \\
\end{tabular}

Do we need to protect \textit{rm}?

\begin{tabular}{p{.5\textwidth}p{.5\textwidth}}
\textbf{We should, but aren’t we late for dinner?} & \textbf{We should, but aren’t we late for dinner?} \\
\end{tabular}

Are you that hungry again?

\begin{tabular}{p{.5\textwidth}p{.5\textwidth}}
\textbf{Try some baba ghanouj followed by moussaka. If that sounds like too much eggplant, escape with a gyro.} & \\
\end{tabular}
Try this hot fudge sundae with coffee ice cream for dessert:

\[
\text{(define } rember1^* \text{)}
\]
\[
\text{(lambda (a l)}
\]
\[
\text{(try}^1 \text{ oh (rm a l oh) l))}
\]

\[\text{1} \text{ Like (and \ldots), (try \ldots) is an abbreviation:}
\]
\[
\text{(try } z \alpha \beta \text{)}
\]
\[
\text{=}
\]
\[
\text{(letcc success}
\]
\[
\text{(letcc} z
\]
\[
\text{(success} \alpha))
\]
\[
\beta\]

The name success must not occur in \(\alpha\) or \(\beta\).

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

We can even simplify \(\text{rm}\) with (try \ldots)

\[
\text{(define } \text{rm}\text{)}
\]
\[
\text{(lambda (a l oh)}
\]
\[
\text{(cond}
\]
\[
\text{((null? } l\text{) (oh (quote no))})
\]
\[
\text{((atom? (car } l\text{))}
\]
\[
\text{(if (eq? (car } l\text{) } a\text{)}
\]
\[
\text{(cdr } l\text{)}
\]
\[
\text{(cons (car } l\text{)}
\]
\[
\text{(rm } a\text{ (cdr } l\text{) oh}))})
\]
\[
\text{(else}
\]
\[
\text{(try } oh2\text{)}
\]
\[
\text{(cons (rm } a\text{ (car } l\text{) } oh2\text{)}
\]
\[
\text{(cdr } l\text{)}
\]
\[
\text{(cons (car } l\text{)}
\]
\[
\text{(rm } a\text{ (cdr } l\text{) oh))})))
\]

Does this version of \(\text{rember1^*}\) rely on \textit{no} being an atom?

No.

Was it a fine dessert?

Yes, but now we are \textit{oh} so very full.
15. The Difference Between Men and Boys...
What is the value of

\[
\text{(define } x \\
\text{ (cons (quote chicago) \\
\text{ (cons (quote pizza) \\
\text{ (quote ()))))})}
\]

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

\[
\text{(set! } x \text{ (quote gone))}
\]

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

\[
\text{(define } x \text{ (quote gone))}
\]

1 L: setq, pronounced “set queue”
S: Pronounced “set bang.”

Did you notice that \text{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

\[
\text{(set! } x \text{ (quote skins))}
\]

Remember this doesn’t have a value.

Is \text{(set! ...)} just like \text{(define ...)}

Yes, mostly.
A \text{(set! ...)} expression always looks like \text{(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

\[
gourmet \quad \text{(define gourmet}
\quad \text{(lambda (food)}
\quad \text{(cons food)
\quad \text{(cons x (quote ()))))}
\]

Now what does \(x\) refer to?

It still refers to skins.

So what is the value of \((\text{cons } x \text{ (quote ())})\)

(skins).

What is the value of \((gourmet \text{ (quote onion)})\)

(onion skins).

\[
\text{(set! } x \text{ (quote rings))}
\]

It is as if we had written:

\[
\text{(define } x \text{ (quote rings))}
\]

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 \((gourmet \text{ (quote onion)})\)

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? No problem. We can use the same trick.

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

(define diner
 (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.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value of ((dinerR \ (quote \ pecanpie)))</td>
<td>(milkshake pecanpie).</td>
</tr>
<tr>
<td>And now what does (x) refer to?</td>
<td>pecanpie.</td>
</tr>
<tr>
<td>Which do you prefer?</td>
<td>Milkshake and pecan pie.</td>
</tr>
<tr>
<td>What is the value of ((gourmand \ (quote \ onion)))</td>
<td>We have done this before:</td>
</tr>
<tr>
<td></td>
<td>(onion onion).</td>
</tr>
<tr>
<td>But, what happened to (x)</td>
<td>It now refers to onion.</td>
</tr>
<tr>
<td>What food did (dinerR) eat last?</td>
<td>Not onion.</td>
</tr>
<tr>
<td>How did that happen?</td>
<td>Both (dinerR) and (gourmand) use (x) to remember the food they saw last.</td>
</tr>
<tr>
<td>Should we have chosen a different name when we wrote (dinerR)?</td>
<td>Yes, we should have chosen a new name.</td>
</tr>
<tr>
<td>Like what?</td>
<td>(y).</td>
</tr>
<tr>
<td>But what would have happened if (gourmand) had used (y) to remember the food it saw last?</td>
<td>Well, wouldn’t we have the same problem again?</td>
</tr>
<tr>
<td>Yes, but don’t worry: there is a way to avoid this conflict of names.</td>
<td>There must be, because we should be able to get around such coincidences!</td>
</tr>
<tr>
<td>Here is a new function:</td>
<td>It looks like (gourmand).</td>
</tr>
<tr>
<td>(\text{define} \ omnivore)</td>
<td>(\text{define omnivore})</td>
</tr>
<tr>
<td>(\text{(let \ (((x \ (quote \ minestrone))))})</td>
<td>(\text{(let ((x (quote minestrone)))})</td>
</tr>
<tr>
<td>(\text{lambda} \ (food))</td>
<td>(\text{lambda (food)})</td>
</tr>
<tr>
<td>(\text{(set!} \ x \ \text{food)})</td>
<td>(\text{(set! x food)})</td>
</tr>
<tr>
<td>(\text{(cons} \ \text{food)})</td>
<td>(\text{(cons food})</td>
</tr>
<tr>
<td>(\text{(cons} \ x)</td>
<td>(\text{(cons x})</td>
</tr>
<tr>
<td>(\text{(quote ())})))</td>
<td>(\text{(quote ())})))}</td>
</tr>
</tbody>
</table>
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?

What is the value of

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

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))

But it really is this:

(define x1 (quote minestrone))

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

Chapter 15
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Did you notice that <code>define</code> is underlined?</td>
<td>Yes, that’s old hat by now.</td>
</tr>
<tr>
<td>Did you see the underlined name?</td>
<td>Yes, and that is something new.</td>
</tr>
<tr>
<td>What is ( x_1 )?</td>
<td>( x_1 ) is an imaginary name.</td>
</tr>
<tr>
<td>Has ( x_1 ) ever been used before with <code>(define ...)</code>?</td>
<td>No, it has not. And it never, ever will be used with <code>(define ...)</code> again.</td>
</tr>
<tr>
<td>What does ( x_1 ) refer to?</td>
<td>It stands for minestrone.</td>
</tr>
<tr>
<td>So, what is ( x_1 )'s value?</td>
<td>No answer; it is imaginary.</td>
</tr>
<tr>
<td>What is the value of <code>omnivore</code></td>
<td>Now it is a function.</td>
</tr>
<tr>
<td>What is the value of <code>(omnivore \( z \))</code> where ( z ) is bouillabaisse</td>
<td>It looks like it is <code>(bouillabaisse bouillabaisse)</code>.</td>
</tr>
<tr>
<td>What is ( x_1 )'s value?</td>
<td>No answer.</td>
</tr>
<tr>
<td>Right?</td>
<td>Always no answer for imaginary names. We just keep in mind what they represent.</td>
</tr>
<tr>
<td>What does ( x_1 ) refer to?</td>
<td>It now stands for bouillabaisse.</td>
</tr>
<tr>
<td>And why?</td>
<td>After determining the value of <code>(omnivore \( z \))</code> where ( z ) is bouillabaisse, ( x_1 ) has changed. It is as if we had written:</td>
</tr>
<tr>
<td></td>
<td><code>(define \( x_1 \) (quote bouillabaisse))</code></td>
</tr>
<tr>
<td></td>
<td>and as if we had never had a definition of ( x_1 ) before.</td>
</tr>
</tbody>
</table>

*The Difference Between Men and Boys...*
Determining the value of \((\text{omnivore } z)\) is just like finding the value of \((\text{gourmand } z)\)

There is no answer for \(z_1\)

Unlike \(x\), \(z_1\) is an imaginary name. We must remember what value it represents, because we cannot find out!

The Sixteenth Commandment

Use \((\text{set! } \ldots)\) only with names defined in \((\text{let } \ldots)\)s.

Take a really close look at this:

This looks like \(\text{omnivore}\).

\[
(\text{define gobbler})
(\text{let } ((x \ (\text{quote minestrone})))
  (\lambda \text{ food})
  (\text{set! } x \text{ food})
  (\text{cons food})
  (\text{cons } x)
  (\text{quote } ()))\]

Not quite. What is the little difference?

The names.

Is there a big difference?

No!

What is the value of

\[
(\text{define gobbler})
(\text{let } ((x \ (\text{quote minestrone})))
  (\lambda \text{ food})
  (\text{set! } x \text{ food})
  (\text{cons food})
  (\text{cons } x)
  (\text{quote } ()))\]

\[
(\text{define } x_2 \ (\text{quote minestrone}))\]

\[
(\text{define gobbler})
(\lambda \text{ food})
(\text{set! } x_2 \text{ food})
(\text{cons food})
(\text{cons } x_2)
(\text{quote } ()))\]
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is $x_2$</td>
<td>$x_2$ is another imaginary name.</td>
</tr>
<tr>
<td>Has $x_2$ ever been used before with (define ...)</td>
<td>No, and it never, ever will be used with (define ...) again.</td>
</tr>
<tr>
<td>What does $x_2$ refer to?</td>
<td>It stands for minestrone.</td>
</tr>
<tr>
<td>What does $x_1$ refer to?</td>
<td>It still stands for bouillabaise.</td>
</tr>
<tr>
<td>So, what is $x_2$'s value?</td>
<td>No answer, because $x_2$ is imaginary.</td>
</tr>
<tr>
<td>What is the value of gobbler</td>
<td>It is a function.</td>
</tr>
<tr>
<td>What is the value of (gobbler $z$) where $z$ is gumbo</td>
<td>It is (gumbo gumbo).</td>
</tr>
<tr>
<td>Now, what is $x_2$'s value?</td>
<td>No answer. Ever!</td>
</tr>
<tr>
<td>What does $x_2$ refer to?</td>
<td>It now stands for gumbo.</td>
</tr>
<tr>
<td>And why?</td>
<td>After determining the value of the definition, the definition of $x_2$ has changed. It is as if we had written:</td>
</tr>
<tr>
<td></td>
<td>(define $x_2$ (quote gumbo))</td>
</tr>
<tr>
<td></td>
<td>and as if we had never had a value for $x_2$ before.</td>
</tr>
</tbody>
</table>
| Determining the value of (gobbler $z$) is just like finding the value of (omnivore $z$) | What is the difference?  

The Difference Between Men and Boys ... 99
There is no answer for $x_2$

Yes, $x_2$ is an imaginary name just as $x_1$ is an imaginary name.

Do omnivore and gobbler observe The Sixteenth Commandment?

They do. The name in (set! ...) is introduced by a (let ...).

What is the value of nibbler

```
(define nibbler
 (lambda (food)
   (let ((x (quote donut)))
     (set! x food)
     (cons food
       (cons x
         (quote ()))))
   ))
```

Unimaginable. Keep reading.

What is the value of (nibbler (quote cheerio))

(cheerio cheerio).

Does nibbler still know about cheerio

No!

How do we determine the value of (nibbler (quote cheerio))

First, we determine the value of (quote donut), which is easy. And then we give it a name: $x$.

And second?

Second, we change what $x$ stands for to cheerio.

And third?

Third, we determine the value of

```
(cons food
  (cons x
    (quote ()))))
```

where $x$ is cheerio and $food$ is cheerio.
So what was the use of (let ...) None. If (let ...) and (set! ...) are used without a (lambda ...) between them, they don’t help us to remember things.

So why is it unimaginable? Because there is no (lambda) between the (let ((x ...)) ...) and the (set! x ...) in (lambda (food)

\[
\ldots
\begin{array}{l}
(\text{let }((x \text{ (quote donut))})) \\
(\text{set! } x \text{ food}) \\
\ldots))
\end{array}
\]

The Seventeenth Commandment

(\textit{preliminary version})

Use (set! x ...) for (let ((x ...)) ...) only if there is at least one (lambda ...) between it and the (let ((x ...)) ...).

Isn’t (let ...) like (letrec ...) Yes, we said it was similar.

Do you think The Sixteenth and Seventeenth Commandments also apply to names in the name part of (letrec ...) Yes, they do, and we will see examples of this, but not just yet.

Why did we forget The Sixteenth Commandment earlier? Occasionally we need to ignore commandments, because it helps to explain things.
Here is the function *glutton*

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

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

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*.”

Explain in your words what it does.

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).
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How can the function change $x$ to what $food$ refers to?</td>
<td>$(\text{set! } x \text{ food})$.</td>
</tr>
<tr>
<td>How many arguments does <em>chez-nous</em> take?</td>
<td>None!</td>
</tr>
<tr>
<td>Is this the right way of putting it all together in one definition?</td>
<td>It is worth a try, but we should check whether it works.</td>
</tr>
<tr>
<td>$(\text{define } \text{chez-nous} \text{ (lambda () \text{(set! food x)} \text{(set! x food))})}$</td>
<td></td>
</tr>
<tr>
<td>What does <em>food</em> refer to?</td>
<td>garlic.</td>
</tr>
<tr>
<td>What does $x$ refer to?</td>
<td>onion.</td>
</tr>
<tr>
<td>What is the value of $(\text{chez-nous})$</td>
<td></td>
</tr>
<tr>
<td>Now, what does <em>food</em> refer to</td>
<td>onion.</td>
</tr>
<tr>
<td>Now, what does $x$ refer to?</td>
<td>onion.</td>
</tr>
<tr>
<td>Did you look closely at the last answer?</td>
<td>We hope so.</td>
</tr>
<tr>
<td>Why is the value of $x$ still onion</td>
<td>After changing <em>food</em> to the value that $x$ stands for, <em>chez-nous</em> changes $x$ to what <em>food</em> refers to.</td>
</tr>
<tr>
<td>And what does <em>food</em> refer to?</td>
<td>onion.</td>
</tr>
</tbody>
</table>
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?

Yes, exactly like that.

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

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
16. Ready, Set, Bang!
Here are *sweet-tooth* and *last*

```scheme
(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 \((\text{sweet-tooth } x)\) where \(x\) is chocolate?

(\(\text{chocolate cake}\)).

What does *last* refer to?

*angelfood*.

What is the value of \((\text{sweet-tooth } x)\) where \(x\) is fruit?

(\(\text{fruit cake}\)).

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:

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

What is the value of \((\text{sweet-toothL } (\text{quote chocolate}))\)

(\(\text{chocolate cake}\)).

And the value of *last* is . . .

*chocolate*.

Ready, Set, Bang!
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value of ((sweet-toothL (quote fruit)))</td>
<td>(fruit cake).</td>
</tr>
<tr>
<td>And last</td>
<td>It refers to fruit.</td>
</tr>
<tr>
<td>Isn't this easy?</td>
<td>Easy as pie!</td>
</tr>
<tr>
<td>Find the value of ((sweet-toothL x)) where (x) is cheese</td>
<td>It is (cheese cake).</td>
</tr>
<tr>
<td>What is the value of ((sweet-toothL (quote carrot)))</td>
<td>(carrot cake).</td>
</tr>
<tr>
<td>Do you still remember the ingredients that went into (sweet-toothL)</td>
<td>There was chocolate, fruit, cheese, and carrot.</td>
</tr>
<tr>
<td>How did you put this list together?</td>
<td>By quickly glancing over the last few questions and answers.</td>
</tr>
<tr>
<td>But couldn't you just as easily have memorized the list as you were reading the questions?</td>
<td>Of course, but why?</td>
</tr>
<tr>
<td>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?</td>
<td>Yes, you can. Here's a hint. [(define ingredients (quote ()))]</td>
</tr>
<tr>
<td>What is that hint about?</td>
<td>This is the name that refers to the list of ingredients that (sweet-toothR) has seen.</td>
</tr>
<tr>
<td>One more hint: The Second Commandment.</td>
<td>Is this the commandment about using (cons) to build lists?</td>
</tr>
</tbody>
</table>
Yes, that's the one. Here's the function:

\[
\text{define } \textit{sweet-toothR} \\
(\text{lambda} \ (\textit{food}) \\
(\text{set! ingredients} \\
(\text{cons food ingredients})) \\
(\text{cons food} \\
(\text{cons (quote cake)} \\
(quote ()))))
\]

What is the value of \(\textit{sweet-toothR } x\) where \(x\) is chocolate (chocolate cake).

What are the \textit{ingredients} (chocolate).

What is the value of \(\textit{sweet-toothR} \ (\text{quote fruit})\) (fruit cake).

Now, what are the \textit{ingredients} (fruit chocolate).

Find the value of \(\textit{sweet-toothR} x\) where \(x\) is cheese It is (cheese cake).

What does the name \textit{ingredients} refer to? (cheese fruit chocolate).

What is the value of \(\textit{sweet-toothR} \ (\text{quote carrot})\) (carrot cake).

And now, what are the \textit{ingredients} (carrot cheese fruit chocolate).

Now that you have had the dessert ... Is it time for the real meal?

Ready, Set, Bang!
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Did we forget about The Sixteenth Commandment?</td>
<td>Sometimes it is easier to explain things when we ignore the commandments. We will use names introduced by (let ...) next time we use (set! ...).</td>
</tr>
<tr>
<td>What is the value of (deep 3)</td>
<td>No, it is not a pizza. It is (((pizza))).</td>
</tr>
<tr>
<td>What is the value of (deep 7)</td>
<td>Don’t get the pizza yet. But, yes, it is ((((((pizza))))))).</td>
</tr>
<tr>
<td>What is the value of (deep 0)</td>
<td>Let’s guess: pizza.</td>
</tr>
<tr>
<td>Good guess.</td>
<td>This is easy: no toppings, plain pizza.</td>
</tr>
<tr>
<td>Is this deep</td>
<td>It would give the right answers.</td>
</tr>
</tbody>
</table>
| (define deep (lambda (m) (cond ((zero? m) (quote pizza)) 
  (else (cons (deep (sub1 m)) (quote ())))))) |                                                                 |
| 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:

```scheme
(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:

```scheme
(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.

```scheme
(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)))`.

---

*Ready, Set, Bang!*
What does \( Ns \) refer to? (3).

And \( Rs \) \(((\text{pizza}))\).

Let's do this again. What is the value of \((\text{deepR } 5)\) \(((\text{pizza}))\).

\( Ns \) refers to ... (5 3).

And \( Rs \) to ... \(((\text{pizza}))\) \(((\text{pizza}))\).

---

**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 \(((\text{pizza}))\).

Now, what does \( Ns \) refer to? (3 5 3).

How about \( Rs \) \(((\text{pizza}))\) \(((\text{pizza}))\) \(((\text{pizza}))\).

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
      ((d (lambda (ns rs)
         (cond
           ((= (car ns) n) (car rs))
           (else
            (A (cdr ns) (cdr rs)))))))
      (A Ns Rs))))
```

No problem.

Use find to write the function deepM which is like deepR but avoids unnecessary consing 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 consed 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 (subl m)) to (deepM (subl m))?  

Do it.

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

What is the value of (deepM 9) (((((((((pizza))))))))).

What is Ns now? (9 8 7 6 5 3).

Ready, Set, Bang!
<table>
<thead>
<tr>
<th><strong>Where did the 7 and 8 come from?</strong></th>
<th>The function <em>deep</em> asks for <em>(deepM 8).</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>And that is why 8 is in the list.</strong></td>
<td><em>(deepM 8)</em> requires the value of <em>(deepM 7).</em></td>
</tr>
<tr>
<td><strong>Is this it?</strong></td>
<td>Yes, because <em>(deepM 6)</em> already knows the answer.</td>
</tr>
<tr>
<td><strong>Can we eat the pizza now?</strong></td>
<td>No, because <em>deepM</em> still disobeys The Sixteenth Commandment.</td>
</tr>
<tr>
<td>**That’s true. The names in <em>(set! Ns ...)</em> and <em>(set! Rs ...)</em> are not introduced by <em>(let ...)</em></td>
<td>It is easy to do that.</td>
</tr>
<tr>
<td><strong>Here it is:</strong></td>
<td>Two imaginary names and <em>deepM.</em></td>
</tr>
<tr>
<td>(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)))))</td>
<td>(define Rs1 (quote ()))</td>
</tr>
<tr>
<td></td>
<td>(define Ns1 (quote ()))</td>
</tr>
<tr>
<td></td>
<td>(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))))</td>
</tr>
<tr>
<td><strong>What is the value of this definition?</strong></td>
<td></td>
</tr>
<tr>
<td><strong>What is the value of <em>(deepM 16)</em></strong></td>
<td>(((((((((((((((((pizza))))))))))))))))).</td>
</tr>
</tbody>
</table>

Chapter 16
Here is what $Ns_1$ refers to:

Our favorite food!

What does $Rs_1$ refer to?

Doesn’t this look like a slice of pizza?

What is $\text{(find 3 (quote () (quote ()) )}$

This questions is meaningless. Neither $Ns_1$ nor $Rs_1$ is empty so $\text{find}$ would never be used like that.

But what would be the result?

No answer.

What would be a good answer?

If $n$ is not in $Ns$, then $(\text{find n Ns Rs})$ should be #f. We just have to add one line to $\text{find}$ if we want to cover this case:

```scheme
(define find
  (lambda (n Ns Rs)
    (letrec
      ((^4 (lambda (ns rs)
          (cond
            {{null? ns) #f)
            ((= (car ns) n) (car rs))
            {else
              (A (cdr ns) (cdr rs))))))))
      (A Ns Rs)))
```

Ready, Set, Bang!
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.

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

(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))))

(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:

(length
 (lambda (l)
  (cond
   ((null? l) 0)
   (else (addl (length (cdr l)))))))
What is the value of

\[
\text{(define length}
  \begin{array}{ll}
  \text{(lambda (l)} & 0) \\
  \end{array}
\)
\]

It is as if we had written:

\[
\text{(define length}
  \begin{array}{ll}
  \text{(lambda (l)} & \\
  \text{(cond} & \\
  \text{((null? l) 0)} & \\
  \text{(else (add1 (length (cdr l))))})) \\
  \end{array}
\)
\]

But doesn’t this disregard The Sixteenth Commandment? Aren’t we supposed to use names in (set!)... that have been introduced by (let ...)?

Here is one way to do it without using a name introduced by (define ...) in a (set! ...)

\[
\text{(define length}
  \begin{array}{ll}
  \text{(let ((h (lambda (l) 0)))} & \\
  \text{(set! h (lambda (l))} & \\
  \text{(cond} & \\
  \text{((null? l) 0)} & \\
  \text{(else (add1 (length (cdr l))))))} & \\
  \end{array}
\)
\]

And this one disregards the The Seventeenth Commandment: there is no (lambda ...) between the (let ((h ...)) ...) and the (set! h ...).

The Seventeenth Commandment

\text{(final version)}

Use (set! x ...) for (let ((x ...)) ...) only if there is at least one (lambda ... between it and the (let ...), or if the new value for x is a function that refers to x.
What is the value of

\[
\begin{align*}
\text{(define } & \text{ length} \\
& \text{(let } ((h \ (\text{lambda} \ (l) \ 0))) \\
& \quad (\text{set! } h \\
& \quad \quad (\text{lambda} \ (l) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad ((\text{null?} \ l) \ 0) \\
& \quad \quad \quad \quad \text{(else } (\text{add1} \ (h \ (\text{cdr} \ l))))))) \\
& \quad h))
\end{align*}
\]

It is as if we had written:

\[
\begin{align*}
\text{(define } & \text{ hi} \\
& \text{(lambda} \ (l) \\
& \quad 0))
\end{align*}
\]

True. Evaluating the definition creates an imaginary definition for \( h \) by removing it from the (let ...)

Yes, and the (let ... ) is now only used to order two events: changing the value of \( h \) and returning the value of \( h_1 \).

What is the value of

\[
\begin{align*}
\text{(define } & \text{ hi} \\
& \text{(lambda} \ (l) \\
& \quad 0))
\end{align*}
\]

\[
\begin{align*}
\text{(define } & \text{ length} \\
& \text{(let} () \\
& \quad (\text{set! } h_1 \\
& \quad \quad (\text{lambda} \ (l) \\
& \quad \quad \quad \text{(cond} \\
& \quad \quad \quad \quad ((\text{null?} \ l) \ 0) \\
& \quad \quad \quad \quad \text{(else } (\text{add1} \ (h_1 \ (\text{cdr} \ l))))))) \\
& \quad h_1))
\end{align*}
\]

It is as if we had written:

\[
\begin{align*}
\text{(define } & \text{ hi} \\
& \text{(lambda} \ (l) \\
& \quad (\text{cond} \\
& \quad \quad ((\text{null?} \ l) \ 0) \\
& \quad \quad \text{(else } (\text{add1} \ (h_1 \ (\text{cdr} \ l))))))))
\end{align*}
\]
What is the value of

\[
\text{(define length}
\begin{array}{c}
\text{(let (l)} \\
\text{h_1))}
\end{array}
\text{)}
\]

It is as if we had written:

\[
\text{(define length}
\begin{array}{c}
\text{(let (l)} \\
\text{h_1))}
\end{array}
\text{)}
\]

Does this mean \textit{length} would perform as we expect it to?

Yes, it would because it is basically the same function it used to be. It just refers to a recursive copy of itself through the imaginary name \textit{h_1}.

Okay, let's start over. Here is the definition of \textit{length} again:

\[
\text{(define length}
\begin{array}{c}
\text{(let ((h (lambda (l) 0)))} \\
\text{(set! h} \\
\text{\quad (lambda (l))} \\
\text{\quad (cond} \\
\text{\quad \quad ((null? l) 0) } \\
\text{\quad \quad (else (addl (h (cdr l)))))}) \\
\text{h))}
\end{array}
\text{)}
\]

The right-hand side of (set! ... ) needs to be eliminated:

\[
\text{(define length}
\begin{array}{c}
\text{(let ((h (lambda (l) 0)))} \\
\text{\quad (set! h ... )} \\
\text{\quad h})
\end{array}
\text{)}
\]

Can you eliminate the parts of the definition that are specific to \textit{length}?

The rest could be reused to construct any recursive function of one argument.

Here is \textit{L}:

\[
\text{(define L}
\begin{array}{c}
\text{(lambda (length)} \\
\text{(lambda (l))} \\
\text{\quad (cond} \\
\text{\quad \quad ((null? l) 0) } \\
\text{\quad \quad (else (addl (length (cdr l)))))})
\end{array}
\text{)}
\]

That should be possible.

Can we use it to express the right-hand side of (set! ... ) in \textit{length}?

Ready, Set, Bang! 121
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is this a good solution?</td>
<td>Yes, except that ((\text{lambda} \ (\text{arg}) \ (h \ \text{arg}))) seems to be a long way of saying h.</td>
</tr>
<tr>
<td>((\text{define} \ \text{length} \ (\text{let} \ (((h \ (\text{lambda} \ (l) \ 0))) \ (\text{set!} \ h \ (L \ (\text{lambda} \ (\text{arg}) \ (h \ \text{arg})))) \ h)))</td>
<td></td>
</tr>
</tbody>
</table>
Rewrite the definition of length so that it becomes a function of $L$. Call the new function $Y_1$.

\[
\text{(define } Y_1 \\
\text{ (lambda } (L) \\
\text{ (let } ((h \text{ (lambda } (l) \text{ (quote } ())))) \\
\text{ (set! } h \\
\text{ (L (lambda } (arg) \text{ (h arg)))) \\
\text{ h)))))
\]

Thank you, Peter J. Landin.

Can you explain $Y$-bang?

\[
\text{(define } Y\text{-bang} \\
\text{ (lambda } (f) \\
\text{ (letrec} \\
\text{ ((h } \text{ (f (lambda } (arg) \text{ (h arg))))) \\
\text{ h))))
\]

Write length using $Y_1$ and $L$.

\[
\text{(define length } (Y_1 \text{ } L))
\]

Here are our words:

"A \text{(letrec } \ldots \text{)} is an abbreviation for an expression consisting of \text{(let } \ldots \text{)} and \text{(set! } \ldots \text{)}. So another way of writing $Y_1$ is $Y$-bang."\(^1\)

\(^1\) A \text{(letrec } \ldots \text{)} that defines mutually recursive definitions can be abbreviated using \text{(let } \ldots \text{)} and \text{(set! } \ldots \text{)} expressions:

\[
\begin{align*}
\text{(letrec} \\
\text{ ((z}_1 \alpha_1) \\
\text{ \ldots} \\
\text{ (z}_n \alpha_n)) \\
\beta
\end{align*}
\]

\[
\text{=}
\begin{align*}
\text{(let } ((y}_1 \alpha_1) \ldots (y}_n \alpha_n)) \\
\text{ (set! } x_1 y_1) \\
\text{ \ldots} \\
\text{ (set! } x_n y_n)) \\
\beta
\end{align*}
\]

The names $y_1 \ldots y_n$ must not occur in $\alpha_1 \ldots \alpha_n$ and they must not be chosen from the names $x_1 \ldots x_n$. Initializing with 0 is arbitrary and it is wrong to assume the names $x_1 \ldots x_n$ are 0 in $\alpha_1 \ldots \alpha_n$.

---

Ready, Set, Bang! 123
You have just worked through the derivation of a function called “the applicative-order, imperative Y combinator.” The interesting aspect of Y is that it produces recursive definitions without requiring that the functions be named by (define ...)

Define D so that depth* is

```
(define depth* (Y D))
```

How do we go from a recursive function definition to a function / such that (Y f) builds the corresponding recursive function without (define ...)

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

Our words:
“f is like the recursive function except that the name of the recursive function is replaced by the name recfun and the whole expression is wrapped in (lambda (recfun) ...).”

Is it true that the value of (Y f) is the same recursive function as the value of (Y f)

Yes, the function Y produces the same recursive function as Y for all f that have this shape.

What happens when we use Y and Y with a function that does not have this shape?

Let’s see.

Give the following function a name:

```
(define ...
  (let ((x 0))
    (lambda (f)
      (set! x (add1 x))
      (lambda (a)
        (if (= a x)
          0
          (f a))))))))
```

How about biz, an abbreviation for bizarre?
That is as good a name as any other. What is the value of this definition?

It is as if we had written:

```
(define x₁ 0)
```

```
(define biz
  (lambda (f)
    (set! x₁ (addl x₁))
    (lambda (a)
      (if (= a x₁)
          0
          (f a))))))
```

What is the value of 

`((Y biz) 5)`

It’s 0.

What is the value of 

`((Y₁ biz) 5)`

It’s not 0. It doesn’t even have an answer!

Does your hat still fit?

Of course it does. After you have worked through the definition of the \(Y\) combinator, nothing will ever affect your hat size again, not even an attempt to understand the difference between \(Y\) and \(Y₁\).

Then again, eating some more scrambled eggs and pancakes may do things to you!

Something lighter, like Belgian waffles, would do it, too.

---

For that elephant ate all night,
And that elephant ate all day;
Do what he could to furnish him food,
The cry was still more hay.

Wang: The Man with an Elephant on His Hands [1891]
—John Cheever Goodwin

Ready, Set, Bang!
17.
We Change, Therefore We Are!
We didn’t expect you so soon.

It is good to be back. What’s next?

Let’s look at deep again.

Here is a definition using (if ...):

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

And let’s look at deepM with the new version of deep included:

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

Can you help D with its work?

Good. Is it true that there is no longer any need for (letrec ...) in deepM

Easy: D should refer to deepM instead of itself.

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

Yes, since D is no longer mentioned in the definition of D.

We Change, Therefore We Are!
This means we can use (let ...)

\[
\begin{align*}
\text{(define } & \text{deepM} \\
\text{ (let } & ((R_s \ \text{quote } ())) \\
& (N_s \ \text{quote } ())) \\
& \text{(let} \\
& ((D \ \text{lambda } (m)) \\
& \quad \text{(if } \text{zero? } m) \\
& \quad \text{(quote } \text{pizza)} \\
& \quad \text{(cons } \text{deepM } (\text{sub1 } m)) \\
& \quad \text{(quote } ()\text{)))) \\
& \text{lambda } (n) \\
& \text{(let } ((\text{exists } (\text{find } n \ N_s R_S))) \\
& \quad \text{(if } \text{atom? } \text{exists}) \\
& \quad \text{(let } ((\text{result } (D n))) \\
& \quad \text{(set! } R_s \ \text{cons result } R_s)) \\
& \quad \text{(set! } N_s \ \text{cons } n \ N_s)) \\
& \quad \text{result} \\
& \text{exists)))))
\end{align*}
\]

Better: there needs to be only one (let ...)

Why?

Because \text{Ns} and \text{Rs} do not appear in the definition of \text{D}

This is true.

\[
\begin{align*}
\text{(define } & \text{deepM} \\
\text{ (let } & ((R_s \ \text{quote } ())) \\
& (N_s \ \text{quote } ())) \\
& (D \ \text{lambda } (m)) \\
& \quad \text{(if } \text{zero? } m) \\
& \quad \text{(quote } \text{pizza)} \\
& \quad \text{(cons } \text{deepM } (\text{sub1 } m)) \\
& \quad \text{(quote } ()\text{)))) \\
& \text{lambda } (n) \\
& \text{(let } ((\text{exists } (\text{find } n \ N_s R_S))) \\
& \quad \text{(if } \text{atom? } \text{exists}) \\
& \quad \text{(let } ((\text{result } (D n))) \\
& \quad \text{(set! } R_s \ \text{cons result } R_s)) \\
& \quad \text{(set! } N_s \ \text{cons } n \ N_s)) \\
& \quad \text{result} \\
& \text{exists)))))
\end{align*}
\]
Can we replace the one use of $D$ by the expression it names?

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

What should we place at the dots?

Since the definition does not contain (set! $D$ ...) and $D$ is used in only one place, we can replace $D$ by its value:

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

What should we place at the dots?

Therefore we can unname an expression that we named with the (let ...)

Yes, that is why the two definitions are equivalent.

Don’t you think applying a (lambda ...) immediately to an argument is equivalent to (let ...)

Yes, determining the value of either one means determining the value of the value parts after associating a name with a value.

Complete the following definition of deepM

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

Is it true that all we got was another (let ...)

And it introduced a name to name another name.
Is there a \( (\text{set! } m \ldots) \) in the value part of \( (\text{let } ((m \ n)) \ldots) \)?

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!

\[
\text{(define } deepM
\text{ (let } ((Rs (quote ())))
\text{ (Ns (quote ())))
\text{ (lambda } (n)
\text{ (let } (((exists (find n Ns RS)))
\text{ (if } (\text{atom? exists})
\text{ (let } (((result \ldots))
\text{ (set! Rs (cons result Rs))
\text{ (set! Ns (cons n Ns))
\text{ result)
\text{ exists))}))))
\text{))})
\]

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 \( \text{conses} \) does \( deep \) use to build \( \text{pizza} \)?

None.

How many \( \text{conses} \) does \( deep \) use to build \( ((((((\text{pizza})))))) \)?

Five, one for each topping.

How many \( \text{conses} \) does \( deep \) use to build \( (((\text{pizza}))) \)?

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 N_1 0)
```

```
(define consC
  (lambda (x y)
    (set! N_1 (add1 N_1))
    (cons x y)))
```

We Change, Therefore We Are!
Could we use this function to determine 500,500?

Sure, no problem.

How?

We just need to use \textit{consC} instead of \textit{cons} in the definition of \textit{deep}:

\begin{verbatim}
(define deep
(lambda (m)
 (if (zero? m)
    (quote pizza)
    (consC (deep (sub1 m))
    (quote ()))))
\end{verbatim}

Wasn’t this exciting?

Well, not really.

So let’s see whether this new \textit{deep} counts \textit{conses}

How about determining the value of \textit{(deep 5)}?

That is easy; we shouldn’t bother. What is the value of \textit{N}_1

We don’t know, it is imaginary.

But that’s how we count \textit{conses}

How could we possibly see something that is imaginary?

Here is one way.

\begin{verbatim}
(define counter)

(define consC
 (let ((N 0))
   (set! counter
     (lambda ()
       N))
  (lambda (x y)
    (set! N (add1 N))
  (cons x y))))
\end{verbatim}

Is this as if we had written:

\begin{verbatim}
(define N_2 0)

(define counter
 (lambda ()
   N_2))

(define consC
 (lambda (x y)
   (set! N_2 (add1 N_2))
  (cons x y)))
\end{verbatim}
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes, what does <em>counter</em> refer to?</td>
<td>A function, perhaps?</td>
</tr>
<tr>
<td>Have we ever seen an incomplete definition before?</td>
<td>No, it looks strange.</td>
</tr>
<tr>
<td><em>(define counter)</em></td>
<td></td>
</tr>
<tr>
<td>It just means that we do not care what the first value of <em>counter</em> is,</td>
<td>... because we immediately change it?</td>
</tr>
<tr>
<td>Correct. But how many arguments does <em>counter</em> take?</td>
<td>None?</td>
</tr>
<tr>
<td>None!</td>
<td>So how do we use it?</td>
</tr>
<tr>
<td>What is the value of <em>(counter)</em></td>
<td>It is whatever $N_2$ refers to.</td>
</tr>
<tr>
<td>And what does $N_2$ refer to?</td>
<td>At this time, 0.</td>
</tr>
<tr>
<td>What is the value of <em>(deep 5)</em></td>
<td>((((((pizza)))))).</td>
</tr>
<tr>
<td>What is the value of <em>(counter)</em></td>
<td>5?</td>
</tr>
<tr>
<td>Yes, 5</td>
<td>How did that happen?</td>
</tr>
<tr>
<td><em>Each time consC</em> is used, one is added to $N_2$. And the answer to <em>(counter)</em> always refers to whatever $N_2$ refers to.*</td>
<td></td>
</tr>
<tr>
<td>What is the value of <em>(deep 7)</em></td>
<td>(((((((pizza))))))).</td>
</tr>
<tr>
<td>What is the value of <em>(counter)</em></td>
<td>Obvious: 12.</td>
</tr>
</tbody>
</table>
Is it clear now how we determine 500,500?

But that is easy. Modify the function supercounter so that it returns the answer of (counter) when it has applied its argument to all the numbers between 0 and 1000

\[
\text{(define supercounter}
\text{(lambda (f))}
\text{(letrec}
\text{((S (lambda (n))}
\text{(if (zero? n))}
\text{(f n))}
\text{(let ()}
\text{(f n))}
\text{(S (sub1 n)))))})
\text{(S 1000)))}
\]

What is the value of \((\text{supercounter f})\) where \(f\) is \textit{deep}?

<table>
<thead>
<tr>
<th>500512.</th>
</tr>
</thead>
</table>

Is this what we expected?

<table>
<thead>
<tr>
<th>No! We wanted 500500.</th>
</tr>
</thead>
</table>

Where did the extra 12 come from?

<table>
<thead>
<tr>
<th>Are these the leftovers from the previous experiments?</th>
</tr>
</thead>
</table>

That's correct.

<table>
<thead>
<tr>
<th>We should not have leftovers.</th>
</tr>
</thead>
</table>

Let's get rid of them.

<table>
<thead>
<tr>
<th>How?</th>
</tr>
</thead>
</table>

Good question! Write a function \textit{set-counter}. What does it do?
The function `set-counter` and `counter` are opposites. Instead of getting the value of the imaginary name, it sets it.

We could modify the definition of `consC`.

```
(define counter)
```

```
(define set-counter)
```

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

And what happens now?

We get three functions and an imaginary name:

```
(define N_3 0)
```

```
(define counter
  (lambda ()
    N_3))
```

```
(define set-counter
  (lambda (x)
    (set! N_3 x)))
```

```
(define consC
  (lambda (x y)
    (set! N_3 (add1 N_3))
    (cons x y)))
```

Now, what is the value of `(set-counter 0)`

We Change, Therefore We Are!
But?  

It changed $N_3$ to 0.

What is the value of $(\text{supercounter } f)$ where $f$ is deep  

500500.

Is this what we expected?  

Yes!

It is time to see how many conses are used for $(\text{deepM 5})$  

Don’t we need to modify its definition so that it uses $\text{consC}$?

Of course! What are you waiting for?

```scheme
(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 conses does $\text{deepM}$ use to build $(((\text{pizza}))))$?  

Probably five?

What is the value of $(\text{counter})$  

500505.

Yes!  

Yes, but it means we forgot to initialize with $\text{set-counter}$.  

Chapter 17
What is the value of \((\text{set-counter} \ 0)\)?

How many \textit{conses} does \textit{deepM} use to build \((((((\text{pizza}))))))

\textbf{Five.}

What is the value of \((\text{counter})\)

\textbf{5.}

What is the value of \((\text{deep} \ 7)\)

\textbf{((((((\text{pizza})))))).}

What is the value of \((\text{counter})\)

\textbf{Obvious: 7.}

Didn’t we need to \textit{set-counter} to 0

\textbf{No, we wanted to count the number of \textit{conses} that were needed to build (\textit{deepM} \ 5) and (\textit{deepM} \ 7).}

Why isn’t this 12

\textbf{Because that was the point of \textit{deepM}.}

\textbf{What is (\textit{supercounter} \(f\)) where \(f\) is \textit{deepM}}

\textbf{Don’t we need to initialize?}

\textbf{No. What is (\textit{supercounter} \(f\)) where \(f\) is \textit{deepM}}

\textbf{1000.}

How many more \textit{conses} does \textit{deep} use to return the same value as \textit{deepM}

\textbf{499,500.}

“A LISP programmer knows the value of everything but the cost of nothing.”

\textbf{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))))))
((((((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)
(more pizza)
(more pizza)
Here is \textit{rember1*} again:

\begin{verbatim}
(define rember1*
  (lambda (a l)
    (letrec
      ((R (lambda (l oh)
            (cond
              ((null? l)
                (oh (quote no)))
              ((atom? (car l))
                (if (eq? (car l) a)
                    (car l)
                    (R (cdr l) oh)))
              (else
                (let ((new-car (letcc oh (R (car l) oh)))
                      (new-l (letcc oh (R (cdr l) oh))))
                  (if (atom? new-car)
                      l
                      new-l))))))

Write it again using our counting version of \textit{cons}

What is the value of (set-counter 0)

What is the value of
\begin{verbatim}
  (rember1*\textit{C} a l)
\end{verbatim}

where
\begin{verbatim}
a is noodles
\end{verbatim}
and
\begin{verbatim}
l is ((food) more (food))
\end{verbatim}

((food) more (food)),

because this list does not contain noodles.

We Change, Therefore We Are!
And what is the value of \((\text{counter})\) \(0\), because we never used \(\text{consC}\). We always used the compass needle and the North Pole to get rid of pending \(\text{consC}\)es.

Do you also remember the first good version of \(\text{rember1*}\)

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

It is the version that failed by repeatedly checking whether anything had changed for the \(\text{car}\) of a list that was a list:

```scheme
(define rember1*C2
 (lambda (a l)
  (letrec
   ((R (lambda (l)
        (cond
          ((null? l) (quote ()))
          ((atom? (car l))
           (if (eq? (car l) a)
                (cdr l)
                (consC (car l)
                        (R (cdr l)))))
          (else
           (let ((av (R (car l))))
             (if (eqlist? (car l) av)
                 (consC (car l)
                         (R (cdr l)))
                 (consC av
                         (cdr l)))))))
   (R l)))))
```

Rewrite it, too, using \(\text{consC}\)

What is the value of \((\text{set-counter} 0)\)

What is the value of

\[
(\text{consC} \ (\text{consC} \ f \ (\text{quote} ()))
 \ (\text{consC} \ m)
 \ (\text{consC} \ (\text{consC} \ f \ (\text{quote} ()))))
\]

where

\(f\) is food

and

\(m\) is more

\(((\text{food}) \ \text{more} \ (\text{food})).\)
What is the value of (counter)

5.

What is the value of (set-counter 0)

\((\text{rember1*}C2 \ a \ l)\)

where

\(a\) is noodles

and

\(l\) is ((food) more (food))

\(((\text{food}) \text{ more } (\text{food}))\),

because this list does not contain noodles.

And what is the value of (counter)

5,

because \(\text{rember1*}C2\) needs five \(\text{cons}\)'s to rebuild the list \(((\text{food}) \text{ more } (\text{food}))\).

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!
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value of <code>(lots 3)</code></td>
<td><code>(egg egg egg)</code></td>
</tr>
<tr>
<td>What is the value of <code>(lots 5)</code></td>
<td><code>(egg egg egg egg egg)</code></td>
</tr>
<tr>
<td>What is the value of <code>(lots 12)</code></td>
<td><code>(egg egg egg egg egg egg egg egg egg egg egg egg)</code></td>
</tr>
<tr>
<td>What is the value of <code>(lenkth (lots 3))</code></td>
<td>3</td>
</tr>
<tr>
<td>What is the value of <code>(lenkth (lots 5))</code></td>
<td>5</td>
</tr>
<tr>
<td>What is the value of <code>(lenkth (lots 15))</code></td>
<td>15</td>
</tr>
<tr>
<td>Here is <code>lots</code></td>
<td>And this is <code>lenkth</code>:</td>
</tr>
<tr>
<td><code>(define lots</code></td>
<td><code>(define lenkth</code></td>
</tr>
<tr>
<td><code>(lambda (m)</code></td>
<td><code>(lambda (l)</code></td>
</tr>
<tr>
<td><code>(cond</code></td>
<td><code>(cond</code></td>
</tr>
<tr>
<td><code>((zero? m) (quote ()))</code></td>
<td><code>((null? l) 0)</code></td>
</tr>
<tr>
<td><code>else </code>(kons(^1)` (quote egg)</td>
<td><code>else (add1 (lenkth (kdr\(^1\) l))))))))</code></td>
</tr>
<tr>
<td><code>(lots (sub1 m))))))))</code></td>
<td></td>
</tr>
</tbody>
</table>

---

1 L, S: This is like `cons`.

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 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 `konsCes` did we use?

The value of `(kounter)` is 3.

Can we add an egg at the end without making any new `konses` except for the last one?

That would be a surprise!
Here is one way.

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

Are there any others?

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)

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

How many konsCes did add-at-end-too use?

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.

1 L: This is like rplacd.
S: This is like set-cdr!.
Recall zubl eddl 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

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

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

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

Write kar and kdr

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

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, kdr₁. And the value that this imaginary name refers to can change over time.

How can we change the value that kdr₁ refers to?

We could write a function that is almost like kar or kdr. This function could use the function (lambda (x) (set! kdr₁ 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 (ad)
    (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 `konses` when we added an egg to the end of the list? Only for the new egg.

What is the value of `dozen`? To find out, we must determine the value of `(lots 12)`.

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

How many `konses` did we use? 12.

What is the value of `bakers-dozen`? To find out, we must determine the value of `(add-at-end dozen)`.

```
(define bakers-dozen (add-at-end dozen))
```
How many konses did we use now? 13.

How many konses did we use altogether? 25.

What is the value of

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

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

How many konses did we use now? One.

How many konses did we use altogether? 26.

Does that mean that the konses in dozen are the same as the first twelve in bakers-dozen-too Absolutely!

Does that mean that the konses in dozen are the same as the first twelve in bakers-dozen Absolutely not!

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

Okay.

How many konses did we use now? 14.

Were you surprised that it wasn’t 13? Yes.

How many konses did we use altogether? 40.

Does that mean that the konses in dozen are the same as the first twelve in bakers-dozen-again Absolutely not, again!
Does that mean that the houses in bakers-dozen are the same as the first twelve in bakers-dozen-again

Absolutely not!

Does that mean that the houses 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)

#t.

where

(define eklist?
  (lambda (Is1 ls2)
    (cond
      ((null? Is1) (null? ls2))
      ((null? ls2) #f)
      (else
       (and (eq? (kar Is1) (kar ls2))
            (eklist? (kdr Is1) (kdr ls2)))))))

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.

We Change, Therefore We Are the Same!
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.

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

(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))))

What is the value of same? bakers-dozen bakers-dozen-too

#t.

Why?

The function same? temporarily changes the kdrs of two konse. 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?

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

How many imaginary names are used to determine the value of same?

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

Here is the function last-kons

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

Describe what it does.

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

(define long (lots 12))

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)))

Still no answer.

What is the value of (lenkth long)
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why is there no value?</td>
<td>Because <code>long</code> is very long.</td>
</tr>
<tr>
<td>How many <code>kons</code> does it contain?</td>
<td>12.</td>
</tr>
<tr>
<td>Didn’t we write <code>length</code> together in <em>The Little Schemer</em>?</td>
<td>Yes, though <code>lenkth</code> now uses <code>kdr</code> because the lists it receives are made with <code>kons</code>.</td>
</tr>
<tr>
<td>Did we disobey any of the commandments when we wrote <code>length</code>?</td>
<td>No, we didn’t!</td>
</tr>
<tr>
<td>Then what’s wrong?</td>
<td>The last <code>kons</code> of <code>long</code> no longer contains <code>(quote ())</code> in the <code>kdr</code> part. Instead, the <code>kdr</code> part refers to some <code>kons</code> inside of <code>long</code>.</td>
</tr>
<tr>
<td>And?</td>
<td>No <code>kdr</code> refers to the empty list, because the only one that did was changed.</td>
</tr>
<tr>
<td>Why is this bad?</td>
<td>It means that <code>lenkth</code> keeps taking <code>kdr</code>s forever.</td>
</tr>
</tbody>
</table>

Draw a picture of "Kons the Magnificent" here.
Here is the function `finite-length` which returns its argument's length, if it has one. If the argument doesn't have a length, the function returns false.

```
(define finite-length
  (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))))))))
```

---

**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 Change, Therefore We Are the Same!*
19.
Absconding with the Jewels
We see you have arrived here. Let’s continue.

What is the value of \((deep\ 6)\)? \((((((pizza))))).\)

Here is \(deep\) again. Yes, this is our friend.

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

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.

Absconding with the Jewels 155
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 ()))))
```

But what if we had started with (deep 4) Then we would have had to define four-layers to create these special pizzas.
Define *four-layers*

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

And how about 1000 layers? Well, we would need to define the function *thousand-layers*. Somehow we seem to define a function that does exactly what is left to do when *deep*'s argument has become 0.

Yes, that’s what we have done. Isn’t there an easier way to do this?

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

Do you mean something like this?

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

That is what we mean. But what do we put where the dots are?

We are about to show you. And how do we make sure the function still returns *pizza* afterward?

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

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

Not even half.

Okay. Let’s see more.

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

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

(define toppings)

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

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

Absconding with the Jewels
Let's add three slices to the mozzarella: 
\[
(\text{cons} \\
(\text{cons} \\
(\text{cons} \text{ (toppings)} \text{ (quote mozzarella)})
(\text{quote}()))
(\text{quote}()))
(\text{quote}()))
\]

\((((((\text{mozzarella})))\)), \text{ 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 \text{(toppings} \text{ m)}\text{ it forgets everything surrounding it and adds exactly six layers of parentheses.}"

Suppose we had started with \text{(deepB 4)}

Then \text{toppings} would be like the function \text{four-layers} but it would still forget.

That means

\[
(\text{cons} \\
(\text{cons} \\
(\text{cons} \text{ (toppings)} \text{ (quote mozzarella)})
(\text{quote}()))
(\text{quote}()))
(\text{quote}()))
\]

\text{would be} \((((((\text{mozzarella}))))\))

Yes!

---

**The Twentieth Commandment**

When thinking about a value created with \text{(letcc \ldots)}, 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

\[
(\text{cons} \text{ (toppings)} \text{ (quote cake)})
(\text{toppings} \text{ (quote cake)})
\]

\((((((\text{cake})))), \text{ no?}\)
Yes, toppings would forget everything. What would be the value of

$$\text{((cons (toppings (quote cake)) (cons (toppings (quote mozzarella)) (cons (toppings (quote pizza)) (quote ()))))}$$

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

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

Does this mean that we can never cons anything onto toppings

Yes, never!

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

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

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

Yes, but collectors are useful here, too.

That's good to know.

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

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

Which function does that?

(lambda (x) x).

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

pizza.
And what is the value of 
\((\text{deep}&\text{co} \ 6 \ (\text{lambda} \ (x) \ x))\) 

\((((((\text{pizza}))))).\)

\((\text{deep}&\text{co} \ 2 \ (\text{lambda} \ (x) \ x))\) 

\(((\text{pizza})), \text{ of course.}\)

And how do we get there?

We ask \((\text{zero}? \ 2)\), which isn’t true, and then determine the value of 
\((\text{deep}&\text{co} \ 1 \ 
(\text{lambda} \ (x) 
\ (k \ (\text{cons} \ x 
\ \ \ \ \ \ (\text{quote} ()))))))\)

where 
\(k\) is \((\text{lambda} \ (x) \ x)\).

How do we do that?

We check whether the first argument is 0 again, and since it still isn’t, we recur with 
\((\text{deep}&\text{co} \ 0 \ 
(\text{lambda} \ (x) 
\ (k \ (\text{cons} \ x 
\ \ \ \ \ \ (\text{quote} ()))))))\)

where 
\(k\) is \((\text{lambda} \ (x) 
(\text{k2} \ (\text{cons} \ x 
\ \ \ \ \ \ (\text{quote} ()))))\))

and 
\(\text{k2}\) is \((\text{lambda} \ (x) \ x)\).

Is there a better way to describe the collector?

Yes, it is equivalent to \textit{two-layers}.

\[
(\text{define} \ \textit{two-layers} \\
(\text{lambda} \ (p) \\
(\text{cons} \\
(\text{cons} \ p \ (\text{quote} ()) \\
(\text{quote} ()))))\)
\]
Why?

We can replace \( k^2 \) with \((\text{lambda} \ (x) \ x)\), which shows that \( k \) is the same as

\[
(\text{lambda} \ (x) \\
(\text{cons} \ x \ (\text{quote} \ ()))).
\]

And then we can replace \( k \) with this new function.

Are we done now?

Yes, we just use \textit{two-layers} on pizza because the first argument is 0, and doing so gives \((\text{pizza})\).

What is the last collector when we determine the value of \((\text{deep}\&\text{co} \ 6 \ (\text{lambda} \ (x) \ x))\)

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

And what is the last collector when we determine the value of \((\text{deep}\&\text{co} \ 4 \ (\text{lambda} \ (x) \ x))\)

\textit{four-layers}.

And now take a close look at the function \textit{deep}\&\textit{coB}

This function remembers the collector in \textit{toppings}.

\[
\begin{align*}
\text{(define} \ \text{deep}\&\text{coB} \\
&\text{(lambda} \ (m \ k) \\
&\text{(cond} \\
&\text{((zero?} \ m) \\
&\text{(let} () \\
&\text{(set!} \ \text{toppings} \ k) \\
&\text{(k} \ (\text{quote} \ \text{pizza})))) \\
&\text{(else} \\
&\text{(deep}\&\text{coB} \ (\text{sub1} \ m) \\
&\text{(lambda} \ (x) \\
&\text{(k} \ (\text{cons} \ x \ (\text{quote} \ ())))))))))
\end{align*}
\]
What is *toppings* after we determine the value of \((\text{deep}\&\text{coB} \ 2 \ (\lambda x \ x))\)  

It is  
\[
(\lambda x \ (k \ (\text{cons} \ x \ (\text{quote} \ ())))))
\]

where  
\[
k \text{ is } (\lambda x \ (k2 \ (\text{cons} \ x \ (\text{quote} \ ()))))
\]
and  
\[
k2 \text{ is } (\lambda x \ x).
\]

So what is it?  
It is *two-layers*.

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

It is equivalent to *six-layers*.

What is the value of \((\text{deep}\&\text{coB} \ 4 \ (\lambda x \ x))\)  

\(((\text{pizza})))\).

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  
\[
(\text{cons} \ (\text{toppings} \ (\text{quote} \ \text{cake})) \ (\text{toppings} \ (\text{quote} \ \text{cake})))
\]

\(((\text{cake}))) \ (((\text{cake}))), \text{ not } (((\text{cake}))))\).

Yes, this version of *toppings* would not forget everything. What would be the value of  
\[
(\text{cons} \ (\text{toppings} \ (\text{quote} \ \text{cake})) \ (\text{cons} \ (\text{toppings} \ (\text{quote} \ \text{mozzarella}))) \ (\text{cons} \ (\text{toppings} \ (\text{quote} \ \text{pizza}))) \ (\text{quote} \ ())))))
\]

\(((\text{cake}))) \ (((\text{mozzarella}))) \ (((\text{pizza})))\).
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))))))
```

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


Easy,

it determines whether any atom occurs twice in a row in a list of atoms.
What is the value of \((\text{two-in-a-row}^*? \ l)\) where
\[ l = ((\text{mozzarella}) (\text{cake}) \text{mozzarella}) \]
Are we going to think about “stars”?

Yes. What is the value of \((\text{two-in-a-row}^*? \ l)\) where
\[ l = ((\text{mozzarella}) (\text{cake}) \text{mozzarella}) \]
\#f.

What is the value of \((\text{two-in-a-row}^*? \ l)\) where
\[ l = ((\text{potato}) (\text{chips ((with) fish) (fish)})) \]
\#t.

What is the value of \((\text{two-in-a-row}^*? \ l)\) where
\[ l = ((\text{potato}) (\text{chips ((with) fish) (chips)})) \]
\#f.

What is the value of \((\text{two-in-a-row}^*? \ l)\) where
\[ l = ((\text{potato}) (\text{chips (chips (with) fish)})) \]
\#t.

Can you explain what \textit{two-in-a-row}^*? does? Here are our words:
“The function \textit{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 \((\text{walk} \ l)\) where
\[ l = ((\text{potato}) (\text{chips (chips (with))) fish}) \]
We haven’t seen \textit{walk} yet.
Here is the definition of \textit{walk}:

\begin{verbatim}
(define leave)

(define walk
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
        (leave (car l)))
      (else
        (let ()
          (walk (car l))
          (walk (cdr l)))))))
\end{verbatim}

Have we seen something like this before?

Yes, \textit{walk} is the minor function \textit{lm} in \textit{leftmost}.

\begin{verbatim}
(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))))
\end{verbatim}

And what does \textit{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

\begin{verbatim}
l is ((potato) (chips (chips (with))) fish)
\end{verbatim}

It uses it on the leftmost atom.

Does this mean \textit{walk} is like \textit{leftmost} if we put the right kind of value into \textit{leave}?

Yes!

What would be the value of (start-it l)

where

\begin{verbatim}
l is ((potato) (chips (chips (with))) fish)
\end{verbatim}

Okay, now \textit{leave} would be a needle!

and the definition for \textit{start-it} is

\begin{verbatim}
(define start-it
  (lambda (l)
    (letcc here
      (set! leave here)
      (walk l))))
\end{verbatim}
<table>
<thead>
<tr>
<th>Why?</th>
<th>Because <code>start-it</code> first sets up a North Pole and then remembers it in <code>leave</code>. When we finally get to <code>(leave (car l))</code>, <code>leave</code> is a needle that is attracted to the North Pole in <code>start-it</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>What would be the value of <code>leave</code></td>
<td>It would be a function that does whatever is left to do after the value of <code>(start-it l)</code> is determined.</td>
</tr>
<tr>
<td>And what would be the value of <code>(start-it l)</code></td>
<td>It would be potato.</td>
</tr>
</tbody>
</table>
| 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))))) |

But?

In addition, waddle would remember rest in fill.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is <em>rest</em></td>
<td>It is a needle, just as <em>jump</em> in <em>deepB</em>.</td>
</tr>
<tr>
<td>Didn’t we say that <em>jump</em> would be like a function?</td>
<td>Yes, it would have been like a function, but when used, it would have also forgotten what to do afterward.</td>
</tr>
<tr>
<td>What kind of function does <em>rest</em> correspond to?</td>
<td>If <em>rest</em> is to <em>waddle</em> what <em>jump</em> is to <em>deepB</em>, the function ignores its argument and then it acts like <em>waddle</em> for the rest of the list until it encounters the next atom.</td>
</tr>
<tr>
<td>Why does this function ignore its argument?</td>
<td>Because the new North Pole creates a function that remembers the rest of what <em>waddle</em> has to do after (<em>letcc</em> ... ) produces a value. Since the value of the first expression in the body of (<em>let</em> () ... ) is ignored, the function throws away the value of the argument.</td>
</tr>
<tr>
<td>What does the function do afterward?</td>
<td>It looks for the first atom in the rest of the list and then uses <em>leave</em> on it. It also remembers what is left to do.</td>
</tr>
</tbody>
</table>
| 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 \textit{rest} \\

No problem:

\begin{verbatim}
(define rest1
  (lambda (x)
    (waddle li)
  ))
\end{verbatim}

where

\textit{li} is (()
  (cheerios (cheerios (spaghettios)))
  donuts).

Was this really no problem?

Well, \textit{x} is never used but that's no problem.

What would be the value of \textit{(get-next (quote go))} where

\begin{verbatim}
(define get-next
  (lambda (x)
    (letcc here-again
      (set! leave here-again)
      (fill (quote go))))
\end{verbatim}

The value would be \textit{cheerios}.

Why?

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

And what else would have happened?

Well, \textit{fill} would now remember a new needle.

And what would this needle correspond to?

It would have corresponded to a function like \textit{rest1}, except that the rest of the list would have been smaller.
Define this function.

\[
\text{(define rest2}
  \quad (\lambda (x)
    \quad (\text{waddle } l2)))
\]

where
\[
l2 \text{ is } (((\text{cheerios } (\text{spaghettios}))) \text{ donuts}).
\]

<table>
<thead>
<tr>
<th>Does \text{get-next} deserve its name?</th>
<th>Yes, it sets up a new North Pole for \text{fill} to return the next atom to.</th>
</tr>
</thead>
<tbody>
<tr>
<td>What else does it do?</td>
<td>Just before \text{fill} determines the next atom in the list of S-expressions that was given to \text{start-it2}, it changes itself so that it can resume the search for the next atom when used again.</td>
</tr>
</tbody>
</table>
| Does this mean that the value of \text{(get-next (quote go))} would be cheerios again? | Yes, if after determining the first value of \text{(get-next (quote go))} we asked for the value again, we would again receive cheerios, because the original list was \((\text{donuts})\)
  \(\quad (\text{cheerios } (\text{cheerios } (\text{spaghettios}))) \text{ donuts})\). |
| And if we were to determine the value of \text{(get-next\textsuperscript{1} (quote go))} a third time, what would we get? | spaghettios, because the next atom in the list is spaghettios. |

\textsuperscript{1} This is not a mathematical function.

Let's imagine we asked \text{(get-next (quote go))} for a fourth time.

Last time: \text{(get-next (quote go))} Wow!

172 Chapter 19
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wow, what?</td>
<td>Since donuts is the very last atom in l, waddle finally reaches (null? l) where l is () .</td>
</tr>
<tr>
<td>And then?</td>
<td>Well, the final value is () .</td>
</tr>
<tr>
<td>What is so bad about that?</td>
<td>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) .</td>
</tr>
<tr>
<td>And from there on?</td>
<td>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.”</td>
</tr>
<tr>
<td>Why would it get back to start-it2</td>
<td>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.</td>
</tr>
<tr>
<td>What should happen instead?</td>
<td>If get-next really deserves its name, it should return (), so that we know that the list is completely exhausted.</td>
</tr>
<tr>
<td>But didn’t we say that get-next deserved its name?</td>
<td>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.</td>
</tr>
<tr>
<td>Does this mean that start-it2 would deserve the name get-first</td>
<td>No, it wouldn’t. It does get the first atom, but later it also returns () when everything is over.</td>
</tr>
</tbody>
</table>
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)

*get-first* l

donut.

(*get-next* (*quote* go))

()

What would (*get-first* l) be where

l was (fish (chips))

fish.
What would be \((\text{get-next} \ (\text{quote go}))\) chips.

What would be \((\text{get-next} \ (\text{quote go}))\) ().

Are there any more atoms to look at? No!

What would \((\text{get-first} \ l)\) be where 
\(l\) is \((\text{fish} \ (\text{chips} \ \text{chips}))\) fish.

What would be \((\text{get-next} \ (\text{quote go}))\) chips.

What would be \((\text{get-next} \ (\text{quote go}))\) chips.

Is it true that chips occurs twice in a row in \((\text{fish} \ (\text{chips} \ \text{chips}))\) Yes, it does! And by using \text{get-first} and \text{get-next}, we can find out!

Should we define \text{two-in-a-row}\* like this: Yes, and here is \text{two-in-a-row-b}\*:

\[
\text{(define two-in-a-row*?)}
\]
\[
\quad \text{(lambda} \ l)\]
\[
\quad \quad \text{(let} ((\text{fst} \ ((\text{get-first} \ l))))\]
\[
\quad \quad \quad \text{(if} (\text{atom?} \ \text{fst})\]
\[
\quad \quad \quad \quad \text{(two-in-a-row-b*?} \ \text{fst})\]
\[
\quad \quad \quad \quad \#f)))\]

\[
\text{(define two-in-a-row-b*?)}
\]
\[
\quad \text{(lambda} \ a)\]
\[
\quad \quad \text{(let} ((\text{n} \ ((\text{get-next} \ (\text{quote go}))))\]
\[
\quad \quad \quad \text{(if} (\text{atom?} \ \text{n})\]
\[
\quad \quad \quad \quad \text{(or} (\text{eq?} \ \text{n} \ a)\]
\[
\quad \quad \quad \quad \quad \quad \text{(two-in-a-row-b*?} \ \text{n})\]
\[
\quad \quad \quad \quad \quad \quad \quad \#f)))\)

Why does \text{two-in-a-row\*} check whether \text{fst} is an atom? Returning (), a non-atom, is \text{get-first}'s way of saying that there is no atom in \(l\).

Why does \text{two-in-a-row-b\*} not take the list as an argument? Because \text{get-next} knows how to get the rest of the atoms, without being told about \(l\).
Why does \textit{two-in-a-row-\(b^*\)}? check whether \(n\) is an atom? Returning (), a non-atom, is \textit{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 \textit{get-first} is only used once, we can get rid of it, too:

\begin{verbatim}
(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))))
\end{verbatim}
Isn't this a large definition?  
Yes, it is. It was a good idea to develop it in several steps.

And what's (two-in-a-row*? l)
where
  l is (((food) ()) (((food))))

#t.

Are you hungry yet?
Very!

Okay, let's hurry then. This is only an appetizer anyway.

What's next?

Real food.

Let's have a banquet.

Hold on!
Why?

Don't forget your banquet, but we still need to do something.

What?

Hop, Skip, and Jump!
20.
What’s in Store?
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

\(\text{(define } x \ 3)\)  

No answer.

What is \(\text{(value } e)\) where \(e\) is \(\text{(define } x \ 3)\)  

No answer.

What is \textit{value}  

The name is familiar from chapter 10. But, the function \textit{value} there does not handle \(\text{(define } \ldots)\).

So the new \textit{value} might be defined like this.

\[
\text{(define value }
  \text{(lambda } (e)
    \ldots
    \text{(cond}
      \text{((define? } e) \ast\text{define } e))
      \text{(else } \text{(the-meaning } e)))
    \ldots)
\]

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 \(\text{(letcc} \ldots)\) now?  

Oh no!

Okay, we’ll wait until later.  

Whew!

Do we need \textit{define}?  

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

\[
\text{(define define?}
  \text{(lambda } (e)
    \text{(cond}
      \text{((atom? } e) \#f)
      \text{((atom? } (\text{car } e))
        \text{(eq? } (\text{car } e) \text{(quote define}))
      \text{(else } \#f)))))
\]
Do we need *`define`?

Yes, we need it. With `define ...`, we can add new definitions.

Here is *`define`:

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

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

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

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

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 *boxes*:

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

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

Does this remind you of something we have discussed before?
Here is a function that changes the contents of a box:

\[
\text{(define setbox}
\begin{array}{l}
\text{(lambda (box new)} \\
\text{(box (lambda (it set) (set new)))))}
\end{array}
\text{)}
\]

Write the function \textit{unbox} which extracts a value from a box:

\[
\text{(define unbox}
\begin{array}{l}
\text{(lambda (box)} \\
\text{(box (lambda (it set) it)))}
\end{array}
\text{)}
\]

That’s easy:

So, is it true that if a name is paired with a boxed value that we can change what the name stands for without changing the table?

Yes, it is. Using \textit{setbox} changes the contents of the box but the table stays the same.

What is the value of \(x\)?

3.

What is \((\text{value e})\) where \(e\) is \(x\)?

3.

Here is \textit{the-meaning}:

\[
\text{(define the-meaning}
\begin{array}{l}
\text{(lambda (e)} \\
\text{(meaning e lookup-in-global-table))})
\end{array}
\text{)}
\]

The function \textit{lookup-in-global-table} is a function that takes a name and looks up its value in \textit{global-table}. It is easy to define:

\[
\text{(define lookup-in-global-table}
\begin{array}{l}
\text{(lambda (name)} \\
\text{(lookup global-table name))})
\end{array}
\text{)}
\]

What do you think \textit{lookup-in-global-table} does?

Is it true that \textit{lookup-in-global-table} is just like a table?

Yes, it is a function that takes a name and returns the value that is paired with the name in \textit{global-table}.

Does this mean \textit{lookup-in-global-table} is like \textit{global-table}?

Yes and no. Since \textit{*define} changes \textit{global-table}, \textit{lookup-in-global-table} is always just like the most recent \textit{global-table}, not like the one we have now.
<table>
<thead>
<tr>
<th>Have we seen this before?</th>
<th>Remember $Y_1$ from chapter 16?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it important that we always have the most recent value of <em>global-table</em></td>
<td>Yes, we will soon see why that is.</td>
</tr>
</tbody>
</table>

Here is *meaning*:

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

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

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

Fine, we will consider it later.

Okay.

Here is the most trivial action.

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

Can you define *identifier*?

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.

```lisp
(define *identifier
  (lambda (e table)
    (unbox (lookup table e))))
```
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value of ( \text{set! } x \ 5 )</td>
<td>No answer.</td>
</tr>
<tr>
<td>What is the value of ( x )</td>
<td>5.</td>
</tr>
<tr>
<td>What is ( \text{value } e ) where ( e ) is ( \text{set! } x \ 5 )</td>
<td>No answer.</td>
</tr>
<tr>
<td>What is ( \text{value } e ) where ( e ) is ( x )</td>
<td>5.</td>
</tr>
<tr>
<td>How does ( *\text{set} ) differ from ( *\text{identifier} )</td>
<td>It too looks up the box that is paired with the name in a (set! ...) expression, but it changes the contents of the box instead of extracting it.</td>
</tr>
<tr>
<td>Where does the new value for the box come from?</td>
<td>It is the value of the right-hand side in a (set! ...) expression.</td>
</tr>
<tr>
<td>Can you write ( *\text{set} ) now?</td>
<td>Yes, it just means translating the words into a definition:</td>
</tr>
</tbody>
</table>
|                                                                         | \[
| (\text{define } *\text{set} \\
| (\lambda (e \ \text{table}) \\
| \text{setbox} \\
| (\text{lookup \ table \ (name-of } e)) \\
| (\text{meaning \ (right-side-of } e \ \text{table})))))
|                                                                         |        |
| Can you describe what \( *\text{set} \) does?                         | Yes. “The function \text{lookup} \ returns the box that is paired with the name whose value is to be changed. The box is then changed so that it contains the value of the right-hand side of the (set! ...) expression.” |
What is the value of \(\text{lambda} (x) \ x\)  
It is a function.

What is \((\text{value } e)\) where  
e is \(\text{lambda} (x) \ x\)  
It could also be a function.

What is the value of  
\(\left(\text{lambda} \ (y)\right)\)  
\(\left(\text{set}! \ x \ 7\right)\)  
\(\ y\)  
\(\ y\)  
\(\ 0\)

What is the value of \(x\)  
7.

What is \((\text{value } e)\) where  
e is \(\left(\text{lambda} \ (y)\right)\)  
\(\left(\text{set}! \ x \ 7\right)\)  
\(\ y\)  
\(\ 0\)  
0.

What is \((\text{value } e)\) where  
e is \(x\)  
0.

What is \((\text{value } e)\) where  
e is \(x\)  
7.

Here is *lambda

```
(define *lambda
  (lambda (e table)
    (lambda (args)
      (beglis (body-of e)
        (multi-extend
          (formals-of e)
          (box-all args)
          table)))
  ))
```

That's interesting, but what are beglis and box-all?
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`?

```
(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.
<table>
<thead>
<tr>
<th>Why did we determine a value that is ignored in the end?</th>
<th>Because the values of all but the last expression in the value part of a \texttt{(lambda ...)} are ignored.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can you summarize now what the function \texttt{beglis} does for \texttt{*lambda}</td>
<td>We summarize: \textquote{The function \texttt{beglis} determines the values of a list of expressions, one at a time, and returns the value of the last one.}</td>
</tr>
<tr>
<td>How does \texttt{*lambda} work?</td>
<td>When given \texttt{(lambda (x y ... ...) ...)}, it returns the function that is in the inner box of \texttt{*lambda}.</td>
</tr>
<tr>
<td>What does that function do?</td>
<td>It takes the values of the arguments and apparently extends \texttt{table}, pairing each formal name, \texttt{x}, \texttt{y}, \ldots, with the corresponding argument value.</td>
</tr>
<tr>
<td>Write the function \texttt{multi-extend}, which takes a list of names, a list of values, and a table and constructs a new table with \texttt{extend}</td>
<td>No problem.</td>
</tr>
</tbody>
</table>
| \(\textbf{(define \texttt{multi-extend}}
\texttt{(lambda (names values table)}
\texttt{\hspace{1em}(cond)}
\texttt{\hspace{2em}((null? names) table)}
\texttt{\hspace{2em}(else)}
\texttt{\hspace{3em}(extend (car names) (car values)}
\texttt{\hspace{4em}(multi-extend)}
\texttt{\hspace{5em}(cdr names)}
\texttt{\hspace{5em}(cdr values)}
\texttt{\hspace{5em}table))))))))) | |
| Okay, so now that we know how \texttt{table} is extended, what happens after the new table is constructed? | The function that represents a \texttt{(lambda ...)} expression uses the resulting table to determine the value of the body of the \texttt{(lambda ...)} expression, which was the first argument to \texttt{*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.
Does this table know about odd?  
It sure does.

Does this table know about even?  
Not yet.

Does this mean that \((\text{value } e)\) where 
\(e\) is \((\text{odd? } 1)\) 
does not have an answer?  
Not yet.

\[(\text{value } e)\] where 
\(e\) is \((\text{define even?})\) 
\[\begin{align*}
\lambda (n) \\
&\quad \begin{cases}
\text{((zero? } n) \#t) \\
\text{(else (odd? (sub1 n)))})
\end{cases}
\end{align*}\]

No answer.

What is \((\text{value } e)\) where 
\(e\) is \((\text{odd? } 1)\) 
\#t. Time for tea and cookies.

Can you explain why?  
Here is how we can explain it: 
"The table that is embedded in the representation of odd? is \(\text{lookup-in-global-table}\). It is like a table, but when it is given a name, it looks in the most current value of \(\text{global-table}\) for the value that goes with the name. Since \(\text{global-table}\) may grow, \(\text{lookup}\) is guaranteed to look through all definitions ever made."

Have we seen this method of changing a function before?  
Yes, when we derived \(Y_i\) in chapter 16, and when we discussed \(\text{lookup-in-global-table}\).

If \(\lambda\) represents \((\lambda \ldots)\) with a function, how does \(\ast\text{application}\) work?  
That is easy. It just applies the value of the first expression in an application to the values of the rest of the application's expressions.
Here is the function *application

\[
\text{define } \ast\text{application} \\
\text{(lambda (e table)} \\
\text{((meaning (function-of e) table)} \\
\text{(evlis (arguments-of e) table))))
\]

The functions function-of and arguments-of are easy ones, and we can write them later. But what does the function evlis do?

\[
\text{define evlis} \\
\text{(lambda (args table)} \\
\text{\textbf{cond} (null? args) (quote ())} \\
\text{\textbf{else} (lambda (val) (cons val (evlis (cdr args) table)) (meaning (car args) table))))
\]

Why do we use \((\text{lambda (val) }\ldots)\ldots\) in evlis

We still don’t have \(\text{let }\ldots\).

Do we need \((\text{lambda (val) }\ldots)\ldots\) here too?

Yes,\(^1\) here and in beglis.

Thank you, John Reynolds.

What happens when we determine the value of \((\text{value e})\)
where
\[e \text{ is (car (cons 0 (quote ()))))}\]

The function value uses the function the-meaning, which in turn uses meaning to determine a value.

And then?

Then expression-to-action determines that \((\text{car (cons 0 (quote ())))}\) is an application, so that \(*\text{application} takes over.

Does this mean the value of
\[(\text{meaning (quote car) table})\]
must be a function?

Yes, because \(*\text{application} expects
\[(\text{function-of e})\text{ to be represented as a (lambda }\ldots\text{)}, \text{ no matter what e is.}
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 ...)

What is (value e) where

\[ e \text{ is (define Is (cons (cons (cons 1 (quote ())) (quote ()) (quote ()))))} \]

We add Is to global-table and remember what it stands for.

What is (value e) where

\[ e \text{ is (car (car (car Is)))} \]
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 \(*\text{const}: (*\text{const} (\text{quote car}))\) tells us.

And that is?  
It is the same as \((a\text{-prim car})\), which is like \(:\text{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.

\((\text{value (quote car)})\)  
We use the function \(*\text{const}: (*\text{const} (\text{quote car}))\) tells us.

And?  
It is the same as \((a\text{-prim car})\), which is like \(:\text{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.

\((\text{value (quote car)})\)  
We use the function \(*\text{const}: (*\text{const} (\text{quote car}))\) tells us.

How often did we have to figure out the value of \((a\text{-prim car})\)  
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 \((\text{let ...})\)  
We can: we just saw how to replace it.

What's in Store?
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
    (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?)))
```

Can you rewrite *const without (let ...)
The Fifteenth Commandment

(final 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. And use (let ...) to name the values of expressions (without set!) that are re-evaluated every time a function is used.

<table>
<thead>
<tr>
<th>Are we now ready to work with value?</th>
<th>Almost.</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is missing?</td>
<td>The one kind of expression that we still need to treat is the set of (cond ...) expressions.</td>
</tr>
<tr>
<td>Is *cond simple?</td>
<td>Yes, there is nothing to it. We must determine the first line in the (cond ...) expression for which the question is true.</td>
</tr>
<tr>
<td>And when we find one?</td>
<td>Then we determine the value of the answer in that line.</td>
</tr>
</tbody>
</table>

Here is the function *cond which uses evcon to do its job:

```
(define *cond
  (lambda (e table)
    (evcon (cond-lines-of e) table)))
```

Can you define the function evcon:

```
(define evcon
  (lambda (lines table)
    (cond
      ((else? (question-of (car lines)))
       (meaning (answer-of (car lines))
                table))
      ((meaning (question-of (car lines))
                table)
       (meaning (answer-of (car lines))
                table))
      (else (evcon (cdr lines) table))))
```

What’s in Store?
What is \((\text{value } e)\) where 
\(e = \text{cond (else 0)}\)

0.

What is \((\text{value } e)\) where 
\(e = \text{cond}
\begin{align*}
  &((\text{null? (cons 0 (quote ()))}) 0) \\
  &\text{(else } 1) \\
\end{align*}

1.

What is \((\text{value } e)\) where 
\(e = \text{cond}\)
No answer.

Time to continue with \((\text{letcc ...})\)
Is it time to go to the North Pole?

Yes, \((\text{letcc skip ...})\) remembers the North Pole so that \text{skip} can find its way back. How does it do this?
We are about to find out.

What does \text{skip} stand for in \((\text{letcc skip ...})\)
We said it was like a function.

Why is it like a function?
We use \((\text{skip } 0)\) when we want to go to the North Pole named \text{skip}.

How is it different from a function?
When we use \text{skip}, it forgets everything that is about to happen.

How can \text{letcc} name a North Pole that remembers what is left to do?
With \((\text{letcc skip ...})\).

And now that \text{skip} is a North Pole, how can we turn it into a function that \text{application} can use?
The North Pole \text{skip} stands for a function of one argument. So the function that represents it for \text{application} must take a list that contains the representation of this argument.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can we use something that we have seen before to make this function?</td>
<td>Yes, we can use ((a\text{-}prim \text{skip})). This is exactly the kind of function we need.</td>
</tr>
<tr>
<td>What is the name for the function just created?</td>
<td>If ((\text{letcc}\text{ skip } \ldots)) is the expression that (*\text{letcc}) receives, then \text{skip} is the name.</td>
</tr>
<tr>
<td>And how do we associate this name with the function we created?</td>
<td>We can use \text{extend} to put the new pair into the table that (*\text{letcc}) receives.</td>
</tr>
<tr>
<td>Here is the function (*\text{letcc})</td>
<td>It sets up the North Pole \text{skip}, turns it into a function that (*\text{application}) can use, associates the name in \text{e} with this function, and evaluates the value part of the expression.</td>
</tr>
<tr>
<td>(\text{(define} *\text{letcc}))</td>
<td></td>
</tr>
<tr>
<td>(\text{(lambda} (e \text{ table})))</td>
<td></td>
</tr>
<tr>
<td>(\text{(letcc} \text{ skip}))</td>
<td></td>
</tr>
<tr>
<td>(\text{(beglis} (\text{ccbody-of} e)))</td>
<td></td>
</tr>
<tr>
<td>(\text{(extend})</td>
<td></td>
</tr>
<tr>
<td>(\text{(name-of} e))</td>
<td></td>
</tr>
<tr>
<td>(\text{(box} (a\text{-}prim \text{skip})))</td>
<td></td>
</tr>
<tr>
<td>(\text{table)}))</td>
<td></td>
</tr>
<tr>
<td>Can you describe what it does?</td>
<td>It sets up the North Pole \text{skip}, turns it into a function that (*\text{application}) can use, associates the name in \text{e} with this function, and evaluates the value part of the expression.</td>
</tr>
<tr>
<td>That's exactly what happens.</td>
<td>Whew.</td>
</tr>
<tr>
<td>But what would happen if we tried to determine the value of ((\text{value} \text{ e})) \text{ where } \text{ e} is \text{ z}</td>
<td>The name \text{z} hasn’t been used with \text{define} yet.</td>
</tr>
<tr>
<td>So what would happen?</td>
<td>We still would like to have a good answer to this question. We have not yet finished the function \text{the-empty-table}.</td>
</tr>
<tr>
<td>Have you forgotten about forgetting? We just showed you how it works.</td>
<td>It is wrong to ask for the value of a name that is not in the table.</td>
</tr>
</tbody>
</table>

*What's in Store?*
What should happen when something wrong happens?

We could forget all pending computations.\(^1\)

\(^1\) We could also use \((\text{letcc} \ldots)\) to remember how the computation would have proceeded, if nothing wrong had happened.

True enough. And how can we forget such pending computations?

We use \((\text{letcc} \ldots)\).

Where should the North Pole be while we determine \((\text{value} e)\)

Right at the beginning of \(\text{value}\):

\[
\begin{aligned}
\text{(define value} \\
\text{(lambda (e)} \\
\text{(letcc the-end)} \\
\ldots \\
\text{(cond)} \\
\text{((define? e) (*define e))} \\
\text{(else (the-meaning e))))})
\end{aligned}
\]

But what can we put in the place of the dots?

Well, we probably should remember \text{the-end} until we are done.

Perhaps we should use \((\text{set!} \ldots)\) to remember it.

Yes, we have always used \((\text{set!} \ldots)\) to remember things.

Here is the final definition of \(\text{value}\)

\[
\begin{aligned}
\text{(define value} \\
\text{(lambda (e)} \\
\text{(letcc the-end)} \\
\text{(set! abort the-end)} \\
\text{(cond)} \\
\text{((define? e) (*define e))} \\
\text{(else (the-meaning e))))})
\end{aligned}
\]

We need to define \text{abort}:

\[
\text{(define abort)}
\]

Can you finish this?
And how does \textit{abort} help us? We should probably use it with 
\textit{the-empty-table}, which is why we redefined \textit{value} in the first place.

Can we now use \textit{abort} inside of \textit{the-empty-table} so that it no longer breaks The Law of Car?

Definitely. Here is how we can fill in the dots in a better way:

\begin{verbatim}
(define the-empty-table
  (lambda (name)
    (abort
      (cons (quote no-answer)
            (cons name (quote ()))))
  
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
        (cond
          ((eq? (car e) (quote quote)) *quote)
          ((eq? (car e) (quote lambda)) *lambda)
          ((eq? (car e) (quote letcc)) *letcc)
          ((eq? (car e) (quote set!)) *set)
          ((eq? (car e) (quote cond)) *cond)
          (else *application))
      (else *application))))
\end{verbatim}

We didn’t talk about \textit{expression-to-action} and \textit{atom-to-action}

\begin{verbatim}
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (else (list-to-action e))))

(define atom-to-action
  (lambda (e)
    (cond
      ((number? e) *const)
      ((eq? e #t) *const)
      ((eq? e #f) *const)
      ((eq? e (quote cons)) *const)
      ((eq? e (quote car)) *const)
      ((eq? e (quote cdr)) *const)
      ((eq? e (quote null?)) *const)
      ((eq? e (quote eq?)) *const)
      ((eq? e (quote atom?)) *const)
      ((eq? e (quote zero?)) *const)
      ((eq? e (quote add1)) *const)
      ((eq? e (quote sub1)) *const)
      ((eq? e (quote number?)) *const)
      (else *identifier)))))
\end{verbatim}

Is there anything left to do?

\begin{verbatim}
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
        (cond
          ((eq? (car e) (quote quote)) *quote)
          ((eq? (car e) (quote lambda)) *lambda)
          ((eq? (car e) (quote letcc)) *letcc)
          ((eq? (car e) (quote set!)) *set)
          ((eq? (car e) (quote cond)) *cond)
          (else *application)))))
\end{verbatim}

Yes, a few simple things:
Here are a few more:

\[
\begin{align*}
&\text{(define text-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(car} \ (cdr \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define formals-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(car} \ (cdr \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define body-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(cdr} \ (cdr \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define ccbody-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(cdr} \ (cdr \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define name-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(car} \ (cdr \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define right-side-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(cond} \\
&\quad \quad \quad \text{((null?} \ (cdr \ (cdr \ x))) \ 0) \\
&\quad \quad \quad \text{else} \ (\text{car} \ (cdr \ (cdr \ x))))))
\end{align*}
\]

\[
\begin{align*}
&\text{(define cond-lines-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(cdr} \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define else?} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(cond} \\
&\quad \quad \quad \text{((atom?} \ x) \ (eq? \ x \ (quote \ else))) \\
&\quad \quad \quad \text{else} \ #f))))
\end{align*}
\]

\[
\begin{align*}
&\text{(define question-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(car} \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define answer-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(car} \ (cdr \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define function-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(car} \ x)))
\end{align*}
\]

\[
\begin{align*}
&\text{(define arguments-of} \\
&\quad \text{(lambda} \ (x) \\
&\quad \quad \text{(cdr} \ x)))
\end{align*}
\]

What is unusual about right-side-of

It returns 0 if there is no right-hand side. This handles definitions like

\[
\text{(define global-table)}
\]

where there is no right-hand side.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How does it take care of such definitions?</td>
<td>It makes up a value for the name until it is changed to what it is supposed to be.</td>
</tr>
<tr>
<td>So what’s the value of all of this?</td>
<td>It makes people hungry.</td>
</tr>
<tr>
<td>What is ((value , e)) where (e) is ((value , 1))</td>
<td>((no-answer , value)).</td>
</tr>
<tr>
<td>How can we teach (value) what value means?</td>
<td>We need to determine the value of ((value , e)) where (e) is ((define , value , (lambda , (e) , (letcc , the-end , (set! , abort , the-end) , (cond , ((define? , e) , (*define , e)) , (else , (the-meaning , e))))))))).</td>
</tr>
<tr>
<td>And then?</td>
<td>Then the answer to our original question is ((no-answer , define?)).</td>
</tr>
<tr>
<td>So we also need to add (define?) to (global-table)</td>
<td>Yes, we do. And while we are at it, we might as well add (*define, the-meaning, lookup, lookup-in-global-table, and a few others.)</td>
</tr>
<tr>
<td>Are you sure we didn’t forget anything?</td>
<td>We can try it out.</td>
</tr>
<tr>
<td>How can we find out what other functions we need?</td>
<td>The same way that we found out that we needed (define?).</td>
</tr>
<tr>
<td>What is ((value , e)) where (e) is ((value , 1))</td>
<td>First we decide that (e) is not a definition, so we determine the value of ((the-meaning , e))).</td>
</tr>
<tr>
<td>And then?</td>
<td>Then we determine the value of (e) (meaning e lookup-in-global-table).</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Is this all?</td>
<td>No. After we find out that (e) is an application, we need to determine (f) (meaning f table) and (a) (meaning a table) where (f) is value, (a) is 1, and table is lookup-in-global-table.</td>
</tr>
<tr>
<td>Is it easy from here on?</td>
<td>The value of value is a function and the value of 1 is 1. The function that represents value extends table by pairing (e) with 1. And now the function works basically like value.</td>
</tr>
<tr>
<td>Does that mean that we get the result 1?</td>
<td>Yes, because we added all the things we needed to global-table.</td>
</tr>
<tr>
<td>If (e) is some expression so that (value (e)) makes sense and if (f) represents (e), then we can always determine the same value by calculating (value value-on-f) where value-on-f is the result of (cons v (cons f (quote ())))) where v is value</td>
<td>That is complicated and true.</td>
</tr>
<tr>
<td>Isn’t it heavy duty work?</td>
<td>It sure burns a lot of calories, but of course that only means that we will soon be ready for a lot more food.</td>
</tr>
</tbody>
</table>
Enjoy yourself with a great dinner:

((escargots garlic)
(chicken Provençal)
((red wine) and Brie))

† No, you don’t have to eat the parentheses.
Welcome to the Show
You have reached the end of your introduction to computation. Are you now ready to tackle a major programming problem? Programming requires two kinds of knowledge: understanding the nature of computation, and discovering the lexicon, features, and idiosyncrasies of a particular programming language. The first of these is the more difficult intellectual task. If you understand the material in *The Little Schemer* and this book, you have mastered that challenge. Still, it would be well worth your time to develop a fuller understanding of all the capabilities in Scheme—this requires getting access to a running Scheme system and mastering those idiosyncrasies. If you want to understand the Scheme programming language in greater depth, take a look at the following books:

**References**


In Fortran you can speak of numbers, and in C of characters and strings. In Lisp, you can speak of Lisp. Everything Lisp does can be described as a Lisp program, simply and concisely. And where shall you go from here? Suppose you were to tinker with the programs in Chapter 20. Add a feature, change a feature . . . You will have a new language, perhaps still like Lisp or perhaps wildly different. The new language may be described in Lisp, yet it will be not Lisp, but a new creation.

If you give someone Fortran, he has Fortran.
If you give someone Lisp, he has any language he pleases.

—Guy L. Steele Jr.
## Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>application</em></td>
<td>190</td>
</tr>
<tr>
<td><em>cond</em></td>
<td>195</td>
</tr>
<tr>
<td><em>const</em></td>
<td>192, 194</td>
</tr>
<tr>
<td><em>define</em></td>
<td>181</td>
</tr>
<tr>
<td><em>identifier</em></td>
<td>183</td>
</tr>
<tr>
<td><em>lambda</em></td>
<td>185</td>
</tr>
<tr>
<td><em>letcc</em></td>
<td>197</td>
</tr>
<tr>
<td><em>quote</em></td>
<td>183</td>
</tr>
<tr>
<td><em>set</em></td>
<td>184</td>
</tr>
<tr>
<td>:car</td>
<td>191</td>
</tr>
<tr>
<td>???, 17</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-prim</td>
<td>191</td>
</tr>
<tr>
<td>abort</td>
<td>198</td>
</tr>
<tr>
<td>add-at-end</td>
<td>144</td>
</tr>
<tr>
<td>add-at-end-too</td>
<td>145</td>
</tr>
<tr>
<td>answer-of</td>
<td>200</td>
</tr>
<tr>
<td>arguments-of</td>
<td>200</td>
</tr>
<tr>
<td>atom-to-action</td>
<td>199</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b-prim</td>
<td>191</td>
</tr>
<tr>
<td>bakers-dozen</td>
<td>147</td>
</tr>
<tr>
<td>bakers-dozen-again</td>
<td>148</td>
</tr>
<tr>
<td>bakers-dozen-too</td>
<td>148</td>
</tr>
<tr>
<td>beglis</td>
<td>186</td>
</tr>
<tr>
<td>biz</td>
<td>125</td>
</tr>
<tr>
<td>body-of</td>
<td>200</td>
</tr>
<tr>
<td>bons</td>
<td>146</td>
</tr>
<tr>
<td>box</td>
<td>181</td>
</tr>
<tr>
<td>box-all</td>
<td>186</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>call-with-current-continuation &amp; <em>letcc</em></td>
<td>41</td>
</tr>
<tr>
<td>chez-nous</td>
<td>103, 104</td>
</tr>
<tr>
<td>cc-body-of</td>
<td>200</td>
</tr>
<tr>
<td>consC</td>
<td>131, 132, 135</td>
</tr>
<tr>
<td>cond-lines-of</td>
<td>200</td>
</tr>
<tr>
<td>counter</td>
<td>132, 133, 135</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>124</td>
</tr>
<tr>
<td>deep</td>
<td>110, 115, 127, 132, 155</td>
</tr>
<tr>
<td>deep&amp;co</td>
<td>161</td>
</tr>
<tr>
<td>deep&amp;coB</td>
<td>163</td>
</tr>
<tr>
<td>deepB</td>
<td>157, 158</td>
</tr>
<tr>
<td>deepM</td>
<td>113, 114, 116, 118, 127–130, 136</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>deepR</td>
<td>111</td>
</tr>
<tr>
<td>define?</td>
<td>180</td>
</tr>
<tr>
<td>depth*</td>
<td>69, 70, 72–75, 122</td>
</tr>
<tr>
<td>diner</td>
<td>94</td>
</tr>
<tr>
<td>dinerR</td>
<td>94</td>
</tr>
<tr>
<td>dozen</td>
<td>147</td>
</tr>
<tr>
<td>eklist?</td>
<td>149</td>
</tr>
<tr>
<td>else?</td>
<td>200</td>
</tr>
<tr>
<td>evcon</td>
<td>195</td>
</tr>
<tr>
<td>even?</td>
<td>188</td>
</tr>
<tr>
<td>eulis</td>
<td>190</td>
</tr>
<tr>
<td>expression-to-action</td>
<td>199</td>
</tr>
<tr>
<td>extend</td>
<td>179</td>
</tr>
<tr>
<td>fill</td>
<td>169</td>
</tr>
<tr>
<td>find</td>
<td>113, 117</td>
</tr>
<tr>
<td>finite-length</td>
<td>153</td>
</tr>
<tr>
<td>food</td>
<td>102</td>
</tr>
<tr>
<td>formals-of</td>
<td>200</td>
</tr>
<tr>
<td>four-layers</td>
<td>157</td>
</tr>
<tr>
<td>function-of</td>
<td>200</td>
</tr>
<tr>
<td>get-first</td>
<td>174</td>
</tr>
<tr>
<td>get-next</td>
<td>171</td>
</tr>
<tr>
<td>global-table</td>
<td>181</td>
</tr>
<tr>
<td>glutton</td>
<td>102</td>
</tr>
<tr>
<td>gobbler</td>
<td>98</td>
</tr>
<tr>
<td>gourmand</td>
<td>93</td>
</tr>
<tr>
<td>gourmet</td>
<td>92</td>
</tr>
<tr>
<td>id</td>
<td>19</td>
</tr>
<tr>
<td>ingredients</td>
<td>108</td>
</tr>
<tr>
<td>intersect</td>
<td>37, 48</td>
</tr>
<tr>
<td>intersectall</td>
<td>38, 39, 41, 49</td>
</tr>
<tr>
<td>is-first-b?</td>
<td>6</td>
</tr>
<tr>
<td>is-first?</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>kar</td>
<td>146</td>
</tr>
<tr>
<td>kdr</td>
<td>146</td>
</tr>
<tr>
<td>kons</td>
<td>146, 147</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>121</td>
</tr>
<tr>
<td>last</td>
<td>107</td>
</tr>
<tr>
<td>last-kons</td>
<td>151</td>
</tr>
</tbody>
</table>
Index

leave, 167
leftmost, 63–66, 76, 78, 81, 82, 167
length, 17, 118–123
length, 143,
letcc & call-with-current-continuation, 41
list-to-action, 199
lm, 78
long, 151
lookup, 179
lookup-in-global-table, 182
lots, 143

max, 75
meaning, 183
member?, 3, 26, 27, 29
mongo, 153
mr, 18
multi-extend, 187
multirember, 17–19, 22, 25, 26
multirember-f, 23, 24

name-of, 200
nibbler, 100
Ns, 111

odd?, 188
omnivore, 95, 96

question-of, 200

pick, 13

rember, 52
rember-beyond-first, 54
rember-eq?, 23
rember-f, 23
rember-upto-last, 57
rember1*, 67, 68, 87, 88, 89, 139, 140
rember1*C, 139
rember1*C2, 140
rest1, 171
rest2, 172

right-side-of, 200
rm, 84, 85, 88, 89
Rs, 111

same?, 150
scramble, 15, 35, 76
scramble-b, 14
set-counter, 135
set-kdr, 147
setbox, 182
six-layers, 156
start-it, 167
start-it2, 169
sum-of-prefixes, 9, 11, 34
sum-of-prefixes-b, 10
supercounter, 134
sweet-tooth, 107
sweet-toothL, 107
sweet-toothR, 109

text-of, 200
the-empty-table, 179, 199
the-meaning, 182
toppings, 158
two-in-a-row*, 175, 176
two-in-a-row-b*, 175
two-in-a-row?, 7, 165
two-in-a-row?, 4–7, 33, 34, 165
two-layers, 162

unbox, 182
union, 27, 28, 31, 32

value, 180, 198

waddle, 169
walk, 167

x, 91, 96, 180
Y-bang, 123
Yl, 123