Library of Congress Cataloging-in-Publication Data

Friedman, Daniel P.
The little LISPper / Daniel P. Friedman, Matthias Felleisen, foreword by Gerald J. Sussman; cover illustrations by Guy L. Steele Jr. -- 3rd ed.
p. cm
Includes index.
ISBN 0-574-24005-5 : $14 00
1. LISP (Computer program language) I. Felleisen, Matthias II. Title
QA76.73.L23F74 1989
005.13'3—dc19

ISBN 0 574-24005-5

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Science Research Associates, Inc.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1
To Mary and Helga,

to our sons

Brian, Robert, and Christopher,

and to the memory of Elliott I. Organick
(Contents
(Foreword ix)
(Preface xi)
((1 Toys) 3)
((2 Do It, Do It Again, and Again, and Again ) 17)
((3 Cons The Magnificent) 37)
((4 Numbers Games) 61)
((5 The Multichapter Chapter) 85)
((6 *Oh My Gawd* It's Full of Stars) 97)
((7 Shadows) 115)
((8 Friends and Relations) 135)
((9 Lambda The Ultimate) 151)
((10 What is the Value of All of This?) 181)
(Welcome to the Show 203)
(Index 205)
)
In 1967 I took an introductory course in photography. Most of the students (including me) came into that course hoping to learn how to be creative—to take pictures like the ones I admired by artists such as Edward Weston. On the first day the teacher patiently explained the long list of technical skills that he was going to teach us during the term. A key was Ansel Adams’ “Zone System” for previsualizing the print values (blackness in the final print) in a photograph and how they derive from the light intensities in the scene. In support of this skill we had to learn the use of exposure meters to measure light intensities and the use of exposure time and development time to control the black level and the contrast in the image. This is in turn supported by even lower level skills such as loading film, developing, and printing, and mixing chemicals. One must learn to ritualize the process of developing sensitive material so that one gets consistent results over many years of work. The first laboratory session was devoted to finding out that developer feels slippery and that fixer smells awful.

But what about creative composition? In order to be creative one must first gain control of the medium. One cannot even begin to think about organizing a great photograph without having the skills to make it happen. In engineering, as in other creative arts, we must learn to do analysis to support our efforts in synthesis. One cannot build a beautiful and functional bridge without a knowledge of steel and dirt and considerable mathematical technique for using this knowledge to compute the properties of structures. Similarly, one cannot build a beautiful computer system without a deep understanding of how to “previsualize” the process generated by the procedures one writes.

Some photographers choose to use black-and-white 8×10 plates while others choose 35mm slides. Each has its advantages and disadvantages. Like photography, programming requires a choice of medium. Lisp is the medium of choice for people who enjoy free style and flexibility. Lisp was initially conceived as a theoretical vehicle for recursion theory and for symbolic algebra. It has developed into a uniquely powerful and flexible family of software development tools, providing wrap-around support for the rapid-prototyping of software systems. As with other languages, Lisp provides the glue for using a vast library of canned parts, produced by members of the user community. In Lisp, procedures are first-class data, to be passed as arguments returned as values and stored in data structures. This flexibility is valuable, but
most importantly, it provides mechanisms for formalizing, naming, and saving the idioms—the common patterns of usage that are essential to engineering design. In addition, Lisp programs can easily manipulate the representations of Lisp programs—a feature that has encouraged the development of a vast structure of program synthesis and analysis tools, such as cross-referencers.

The Little LISPer is a unique approach to developing the skills underlying creative programming in Lisp. It painlessly packages, with considerable wit, much of the drill and practice that is necessary to learn the skills of constructing recursive processes and manipulating recursive data-structures. For the student of Lisp programming, The Little LISPer can perform the same service that Hanon’s finger exercises or Czerny’s piano studies perform for the student of piano.

Gerald J. Sussman
Cambridge, Massachusetts
Preface

Recursion is the act of defining an object or solving a problem in terms of itself. A careless recursion can lead to an infinite regress. We avoid the bottomless circularity inherent in this tactic by demanding that the recursion be stated in terms of some “simpler” object, and by providing the definition or solution of some trivial base case. Properly used, recursion is a powerful problem solving technique, both in artificial domains like mathematics and computer programming, and in real life.

The goal of this book is to teach the reader to think recursively. Our first task, therefore, 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 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 entire technical content of this book in less than a page of mathematics, but the reader who understands that page has little need for this book. For most people, formal mathematics is not very intuitive. The marriage of technology and mathematics presents us with a third, almost ideal choice: a programming language. Programming languages are perhaps the best way to convey the concept of recursion. 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 and try them, observe their behavior, modify them, and experience the effect of your modifications.

Perhaps the best programming language for teaching recursion is Lisp. Lisp is inherently symbolic—the programmer does not have to make an explicit mapping between the symbols of his own language and the representations in the computer. Recursion is Lisp’s natural computational mechanism; the primary programming activity is the creation of (potentially) recursive definitions. Lisp implementations are predominantly interactive—the programmer can immediately participate in and observe the behavior of his programs. And, perhaps most
importantly for our lessons at the end of this book, there is a direct correspondence between
the structure of Lisp programs and the data those programs manipulate

Lisp is practical. It is the dominant language for work in artificial intelligence: computational linguistics, robotics, pattern recognition, expert systems, generalized problem solving, theorem proving, game playing, algebraic manipulation, etc. It has had a major influence on
most other fields of computer science

Although Lisp can be described quite formally, understanding Lisp does not require a
particularly mathematical inclination. In fact, The Little LISPer is based on lecture notes
from a two-week "quickie" introduction to Lisp for students with no previous programming
experience and an admitted dislike for anything quantitative. Many of these students were
preparing for careers in public affairs. It is our belief that writing programs recursively in Lisp
is essentially simple pattern recognition. Since our only concern is recursive programming, our
treatment is limited to the why’s and wherefore’s of just a few Lisp features: car, cdr, cons,
eq?, atom?, null?, number?, zero?, add1, sub1, not, and, or, quote, lambda, define, and cond.
Indeed, our language is an idealized Lisp.

The Little LISPer is not a complete book on Lisp. However, mastery of the concepts in
this book is mastery of the foundations of Lisp—after you understand this material, the rest
will be easy

Acknowledgements

Many people made important contributions to the first edition of this book. The following
acknowledgement appeared there:

Many thanks to John McCarthy, Mike Greenawalt, John Howard, Terry Pratt, David
Musser, William Gear, Mark Elson, Harold Stone, Jonathan Slocum, Juny Armus, Bob
Wesson, and Marylin Sealy for their extremely helpful comments about The Little LISPer.
Grateful appreciation to Ben Shneiderman and Steve Mitchell for believing in this un-
orthodox approach to Lisp. Thank you to Bill Cohagan who taught me Lisp and to Ann
and Frances Patterson for their careful typing and proofreading of this manuscript. I am
greatly indebted to Patrick Mahaffey for his futuristic ideas of teaching Lisp to public
affairs graduate students and to John Gronouski, Dean of the Lyndon Baines Johnson
School of Public Affairs, who was persuaded to try the experiment. I want to thank
the four public affairs graduate students, Phillip S. Blackerby, Robert J. King, David L Wal-
rath, and Abraham Goldberg who not only were students in the "quickie" course but also
persevered with me an additional week for the completion of the first draft. I especially
want to thank Abraham Goldberg who contributed his editing talents and novice per-
spective for that week. Above all, I want to thank my wife Mary for her understanding
encouragement, and concern for my well-being during the completion of this manuscript.

We are indebted to many people for their contributions and assistance throughout the develop-
ment of this book. We thank Bruce Duba, Kent Dybvig, Chris Haynes, Eugene Kohlbecker,
Richard Salter, George Sprunger, Mitch Wand, and David S. Wise for countless discussions
which influenced our thinking while conceiving this book. Ghassan Abbas, Charles Baker,
David Boyer, Mike Dunn, Terry Falkenberg, Robert Friedman, John Gateley, Mayer Goldberg,
Iqbal Khan, Julia Lawall, Jon Mendelsohn, John Nienart, Jeffrey D. Perotti, Ed Robertson,
Guidelines for the Reader

Do not rush through this book. Read carefully; valuable hints are scattered throughout the text. Do not read the book in less than three sittings unless you are already familiar with Lisp but are not a "LISPer." 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.

*Guess!* This book is based on intuition, and yours is as good as anyone's. Also, if you can, try the examples while you read. Lisps are readily available. While there are minor syntactic variations between different implementations of Lisp (primarily the spelling of particular names and the domain of specific functions), Lisp is basically the same throughout the world. To work with Lisp, you may need to modify the programs slightly. Typically, the material requires only a few changes for modern Lisps such as COMMON Lisp [4] and Scheme [1, 2]. Suggestions about how to try the programs in the book are provided in the footnotes. Footnotes preceded by "L:" concern Lisp, those by "S:" concern Scheme. For Scheme, you may have to enter the definitions of add1, sub1, and atom? because some implementations do not provide these functions:

```scheme
(define add1 (let ((+)) (lambda (x) (+ x 1))))
(define sub1 (let ((-)) (lambda (x) (- x 1))))
(define atom? (let ((pair?) (not)) (lambda (x) (not (pair? x)))))
```

We have formulated these definitions in such a way that they are safe from re-definition of built-in functions, this is particularly important for Chapter 4 where we discuss versions of + and - in terms of add1 and sub1.

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 *Laws* and *Commandments* thoroughly before passing them by. The key to learning Lisp is "pattern recognition." The *Commandments* point out the patterns that you will have already seen. 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 Lisp, Lisp itself is more general and incorporates more than we could intelligibly cover in an introductory text. After you have mastered this book, you can read and understand more advanced and comprehensive books on Lisp.

We use a few notational conventions throughout the text, primarily changes in font for different classes of symbols. Programs in notes preceded by "L:" or "S:" are set in typewriter font. Function definitions are in roman characters, parameters are in *italic*, and data
is in sans serif. The values for true and false are in slanted font. Special symbols such as define and cond are in boldface. These distinctions can be safely ignored until Chapter 10, when we treat programs as data. We have taken certain liberties with punctuation to increase clarity.

Food appears in many of our examples for two reasons. First, food is easier to visualize than abstract symbols. (This is not a good book to read while dieting.) We hope the choice of food will help you understand the examples and concepts we use. Second, we want to provide you with a little distraction. We know how frustrating the subject matter can be, and a little distraction will help you keep your sanity.

You are now ready to start. Good luck! We hope you will enjoy the challenges waiting for you on the following pages.

Bon appetit!

Daniel P. Friedman
Matthias Felleisen
Bloomington, Indiana
About the cover

Recursion Yin and Langda

The ancient and beautiful Chinese yin-yang symbol represents complementary but harmonious principles (such as earth and heaven, female and male, dark and light) that make up all aspects of life and the universe.

In modified form it illustrates the complementary but harmonious relationship between two important theories of recursive functions: the combinatory calculus and the lambda calculus. Each separately is computationally universal, but the combinatory calculus has no variables while the very essence of the lambda calculus (on which Lisp is based) is the correct treatment of bound variables—yet the two calculi can be integrated smoothly.

By a curious coincidence, the symbols most representative of the two calculi, Y and λ, are related by a 180° rotation.

Illustration and commentary by Guy L Steele Jr
Toys
Is it true that this is an atom?  

1. Yes, because atom is a string of characters beginning with the letter a

   L.S: (quote atom) or 'atom  
   "L." and "S." are described on page xii

Is it true that this is an atom?  

turkey

2. Yes, because turkey is a string of characters beginning with a letter

Is it true that this is an atom?  

1492

3. Yes, since 1492 is a string of characters beginning with a digit.

Is it true that this is an atom?  

3turkeys

4. Yes, since 3turkeys is a string of characters beginning with a digit

Is it true that this is an atom?  

u

5. Yes, because u is a string of one character beginning with a letter or digit.

Is it true that this is an atom?  

*abc$

6. Yes, because *abc$ is a string of characters beginning with a letter, digit, or special character other than a left "(" or right ")" parenthesis

Is it true that this is a list?  

1. Yes, because (atom) is an atom enclosed by parentheses
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it true that this is a list? (atom turkey or)</td>
<td>Yes, because it is a collection of atoms enclosed by parentheses</td>
</tr>
<tr>
<td>Is it true that this is a list? (atom turkey) or</td>
<td>No, since this is actually two S expressions not enclosed by parentheses. The first one is a list containing two atoms, and the second one is an atom</td>
</tr>
<tr>
<td>Is it true that this is a list? ((atom turkey) or)</td>
<td>Yes, because the two S-expressions are now enclosed by parentheses.</td>
</tr>
<tr>
<td>Is it true that this is an S-expression? xyz</td>
<td>Yes, because all atoms are S expressions</td>
</tr>
<tr>
<td>Is it true that this is an S expression? (xyz)</td>
<td>Yes, because it is a list</td>
</tr>
<tr>
<td>Is it true that this is an S expression? ((xyz) z)</td>
<td>Yes, because all lists are S expressions</td>
</tr>
<tr>
<td>Is it true that this is a list? (how are you doing so far)</td>
<td>Yes, because it is a collection of S expressions enclosed by parentheses</td>
</tr>
<tr>
<td>How many S-expressions are in the list (how are you doing so far) and what are they?</td>
<td>Six, how, are, you doing, so, and far</td>
</tr>
<tr>
<td>Question</td>
<td>Answer</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Is it true that this is a list?</td>
<td>Yes, because it is a collection of S expressions enclosed by parentheses</td>
</tr>
<tr>
<td>(((how) are) ((you) (doing so)) far)</td>
<td></td>
</tr>
<tr>
<td>How many S-expressions are in the list</td>
<td>Three, (((how) are), ((you) (doing so)), and far</td>
</tr>
<tr>
<td>and what are they?</td>
<td></td>
</tr>
<tr>
<td>Is it true that this is a list?</td>
<td>Yes, because it contains zero S expressions enclosed by parentheses. This special S expression is called the null list</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>Is it true that this is an atom?</td>
<td>Yes, because () is both a list and an atom</td>
</tr>
<tr>
<td>()</td>
<td></td>
</tr>
<tr>
<td>Is it true that this is a list?</td>
<td>Yes, because it is a collection of S expressions enclosed by parentheses</td>
</tr>
<tr>
<td>( () () () () )</td>
<td></td>
</tr>
<tr>
<td>What is the car of $l$, where $l$ is the argument $a$ b c</td>
<td>$a$, because $a$ is the first atom of this list</td>
</tr>
<tr>
<td>(a b c)</td>
<td></td>
</tr>
<tr>
<td>What is the car of $l$, where $l$ is the argument ((a b c) x y z)</td>
<td>(a b c), because (a b c) is the first S expression of this non null list</td>
</tr>
<tr>
<td>((a b c) x y z)</td>
<td></td>
</tr>
<tr>
<td>What is the car of $l$, where $l$ is the argument hotdog</td>
<td>No answer, you cannot ask for the car of an atom</td>
</tr>
</tbody>
</table>
The Law of Car

Car is defined only for non-null lists.

What is the car of $l$, where $l$ is the argument

$(((\text{hotdogs})) \ (\text{and}) \ (\text{pickle}) \ \text{relish})$

$(((\text{hotdogs}))$.

Read as:

"The list of the list of hotdogs."

$(((\text{hotdogs}))$ is the first S-expression of $l$

What is (car $l$), where $l$ is the argument

$(((\text{hotdogs})) \ (\text{and}) \ (\text{pickle}) \ \text{relish})$

$(((\text{hotdogs}))$,

because (car $l$) is another way to ask for

"the car of the list $l$."

What is (car (car $l$)), where $l$ is the argument

$(((\text{hotdogs})) \ (\text{and}))$

(hotdogs)

What is the $cdr$ of $l$, where $l$ is the argument

(a b c)

(b c),

because (b c) is the list $l$, without (car $l$)

Note "$cdr$" is pronounced 'could er"

What is the $cdr$ of $l$, where $l$ is the argument

$((a \ b \ c) \ x \ y \ z)$

(x y z)
What is \((\text{cdr } a)\), where \(a\) is the argument

<table>
<thead>
<tr>
<th>Hotdogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>No answer</td>
</tr>
<tr>
<td>You cannot ask for the cdr of an atom</td>
</tr>
</tbody>
</table>

What is \((\text{cdr } l)\), where \(l\) is the argument

| () |
| No answer |
| You cannot ask for the cdr of the null list |

---

**The Law of Cdr**

Cdr is defined only for non-null lists. The cdr of any non-null list is always another list.

---

What is \((\text{car } (\text{cdr } l))\), where \(l\) is the argument

<table>
<thead>
<tr>
<th>((b) (x y) ((c)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((x y)), because (((x y) ((c)))) is ((\text{cdr } l)), and ((x y)) is the car of ((\text{cdr } l)).</td>
</tr>
</tbody>
</table>

What is \((\text{cdr } (\text{cdr } l))\), where \(l\) is the argument

<table>
<thead>
<tr>
<th>(((b)) (x y) ((c)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(((c))), because (((x y) ((c)))) is ((\text{cdr } l)), and (((c))) is the cdr of ((\text{cdr } l))</td>
</tr>
</tbody>
</table>

What is \((\text{cdr } (\text{car } l))\), where \(l\) is the argument

<table>
<thead>
<tr>
<th>((a) (b (c)) d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No answer, since ((\text{car } l)) is an atom, and cdr does not take an atom for an argument; see The Law of Cdr.</td>
</tr>
</tbody>
</table>
What does car take as an argument? It takes any non null list as its argument

What does cdr take as an argument? It takes any non null list as its argument

What is the cons of the atom a and the list l, where
a is the argument peanut, and
l is the argument (butter and jelly)
This can also be written "(cons a l)."
Read: "cons the atom a onto the list l"

What is the cons of s and l, where
s is (mayonnaise and), and
l is (peanut butter and jelly)

What is (cons s l), where
s is ((help) this), and
l is (is very ((hard) to learn))

What does cons take as its arguments?
cons takes two arguments
the first one is any S expression,
the second one is any list.

What is (cons s l), where
s is (a b (c)), and
l is ()

What is (cons s l) where
s is a, and
l is ()

((peanut butter and jelly),
because cons adds an atom to the front of a list

((mayonnaise and) peanut butter and jelly)
because cons adds any S expression to the front of a list

(((help) this) is very ((hard) to learn))

((a b (c))),
because () is a list

(a)
What is (cons \(s\) \(l\)), where
\(s\) is \(a\), and
\(l\) is \(b\)

No answer
Why?

---

**The Law of Cons**

Cons takes two arguments. The second argument of cons must be a list. The result is a list.

---

What is (cons \(s\) (car \(l\))), where
\(s\) is \(a\), and
\(l\) is ((b) c d)

(a b)
Why?

---

What is (cons \(s\) (cdr \(l\))), where
\(s\) is \(a\), and
\(l\) is ((b) c d)

(a c d)
Why?

---

Is it true that the list \(l\) is the null list, where
\(l\) is ( )

Yes,
because it is the list composed of zero S expressions

This question can also be written
(null? \(l\))
Is (null? l) true or false, where l is the argument (a b c)  
False, because it is a non null list

Is (null? a) true or false, where a is spaghetti  
No answer, because you cannot ask null? of a non null atom

--

1 In practice, (null? a), where a is an atom, is well defined. The beginner should follow The Law of Null?

---

The Law of Null?

Null? is defined only for lists.

---

Is it true or false, that s is an atom, where s is Harry  
True, because Harry is a string of characters beginning with a letter

Is (atom? s) true or false, where s is Harry  
True, because (atom? s) is just another way to ask, "Is it true or false that s is an atom?"

---

1 L: atom  
S: See Preface page xiii
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
</table>
| Is \((\text{atom? } s)\) true or false, where 
  \(s\) is \((\text{Harry had a heap of apples})\) | False, since the argument \(s\) is a list |
| How many arguments does \(\text{atom?}\) take, and what are they?       | It takes one argument. The argument can be any S-expression |
| Is \((\text{atom? } (\text{car } l))\) true or false, where 
  \(l\) is \((\text{Harry had a heap of apples})\) | True, because \((\text{car } l)\) is Harry, and Harry is an atom |
| Is \((\text{atom? } (\text{cdr } l))\) true or false, where 
  \(l\) is \((\text{Harry had a heap of apples})\) | False |
| Is \((\text{atom? } (\text{cdr } l))\) true or false where 
  \(l\) is \((\text{Harry})\) | True, because the list \((\ )\) is also an atom |
| Is \((\text{atom? } (\text{car } (\text{cdr } l)))\) true or false, where 
  \(l\) is \((\text{swing low sweet cherry})\) | True, because \((\text{cdr } l)\) is \((\text{low sweet cherry})\), and \((\text{car } (\text{cdr } l))\) is low, which is an atom |
| Is \((\text{atom? } (\text{car } (\text{cdr } l)))\) true or false, where 
  \(l\) is \((\text{swing (low sweet) cherry})\) | False, since \((\text{cdr } l)\) is \((\text{(low sweet) cherry})\), and \((\text{car } (\text{cdr } l))\) is \((\text{low sweet})\), which is a bst |
| True or false \(a1\) and \(a2\) are the same atom, where 
  \(a1\) is Harry, and \(a2\) is Harry | True, because \(a1\) is the atom Harry and \(a2\) is the atom Harry |
Is (eq? a1 a2) true or false, where
   a1 is margarine, and
   a2 is butter

False,
   since the arguments a1 and a2 are different atoms

How many arguments does eq? take and what are they?

It takes two arguments. Both of them must be atoms

Is (eq? l1 l2) true or false, where
   l1 is () and
   l2 is (strawberry)

No answer,¹
   although () is an atom, (strawberry) is a non-null list

¹ Lists may be arguments of eq? Two lists are eq? if they are the same list. Two lists that print the same are equal?, but they are not necessarily eq? The beginner should follow The Law of Eq?

The Law of Eq?

Eq? takes two arguments. Each must be an atom.

Is (eq? (car l) a) true or false, where
   l is (Mary had a little lamb chop), and
   a is Mary

True,
   because (car l) is the atom Mary, and the argument a is also the atom Mary
Is `(eq? (cdr l) a)` true or false, where
l is (soured milk), and
a is milk

No answer.
See The Laws of Cdr and Eq?

Is `(eq? (car l) (car (cdr l)))` true or false,
where
l is (beans beans we need jelly beans)

True,
as this compares the first and second
atoms in the list.

⇒ Now go make yourself a peanut butter and jelly sandwich. ⇐

This space reserved for

JELLY STAINS!
Exercises

1.1 Think of ten different atoms and write them down.

1.2 Using the atoms of Exercise 1.1, make up twenty different lists.

1.3 The list (all these problems) can be constructed by \((\text{cons } a \text{ cons } b \text{ cons } c \text{ a is all, b is these, c is problems, and d is } (\text{ )))\)

Write down how you would construct the following lists:
- \((\text{all these problems}))\)
- \((\text{all these problems}))\)
- \((\text{all these problems}))\)
- \((\text{all these problems}))\)

1.4 What is \((\text{car } (\text{cons } a \text{ l}))\), where \(a\) is french, and \(l\) is \((\text{fries})\), and what is \((\text{cdr } (\text{cons } a \text{ l}))\), where \(a\) is oranges, and \(l\) is \((\text{apples and peaches})\)?

1.5 Find an atom \(x\) that makes \((\text{eq? } x \text{ y})\) true, where \(y\) is lisp. Are there any?

1.6 If \(a\) is atom, is there a list \(l\) that makes \((\text{null? } (\text{cons } a \text{ l}))\) true?

1.7 Determine the value of
- \((\text{cons } s \text{ l})\), where \(s\) is \(x\), and \(l\) is \(y\)
- \((\text{cons } s \text{ l})\), where \(s\) is \((\text{ })\), and \(l\) is \((\text{ })\)
- \((\text{car } s)\), where \(s\) is \((\text{ })\)
- \((\text{cdr } l)\), where \(l\) is \((\text{ (( )}))\)
True or false,

\[(\text{atom? (car } l\)), \text{ where } l \text{ is ((meatballs) and spaghetti)}\]

\[(\text{null? (cdr } l\)), \text{ where } l \text{ is ((meatballs)}\]

\[(\text{eq? (car } l\) (\text{car (cdr } l\))\), \text{ where } l \text{ is (two meatballs)}\]

\[(\text{atom? (cons a } l\)), \text{ where } l \text{ is (ball) and a is meat}\]

What is

\[(\text{car (cdr (cdr (car } l\))})\) \text{ where } l \text{ is ((kiwis mangoes lemons) and (more)}\]

\[(\text{car (cdr (car (cdr } l\))})\) \text{ where } l \text{ is (( } (\text{eggs and (bacon)}) (for) (breakfa}\]

\[(\text{car (cdr (cdr (cdr } l\))})\) \text{ where } l \text{ is ((( } ( ) ( ) ( (and (coffee)) please)}\]

To get the atom and in (peanut butter and jelly on toast) we can write (car t would you write to get.

\[\text{Harry in } l, \text{ where } l \text{ is (apples in (Harry has a backyard)\)]

\[\text{where } l \text{ is (apples and Harry)\]

\[\text{where } l \text{ is (((apples) and ((Harry))) in his backyard)}\]

Do It, Do It Again, and Again, and Again...
| True or false: (lat? l), where | True, because each S expression in l is an atom |
| l is (Jack Sprat could eat no chicken fat) | |
| False, since (car l) is a list | True or false: (lat? l), where |
| l is ((Jack) Sprat could eat no chicken fat) | False, since one of the S expressions in l is a list |
| True or false (lat? l), where | True, because ( ) contains no lists and because it does not contain any lists, it is a lat |
| l is ( ) | True or false a lat is a list of atoms Every lat is a list of atoms |
| Write the function lat? using some, but not necessarily all, of the following functions: car, cdr, cons null? atom? and eq? | We did not expect you to know this, because you are still missing some ingredients. Go on to the next question. Good luck |

\[
\text{(define} \ t^1 \text{ lat?)} \\
\quad \text{(lambda} \ l) \\
\quad \quad \text{(cond}} \\
\quad \quad \quad \quad ((\text{null?} \ l) \ t) \\
\quad \quad \quad \quad ((\text{atom?} \ (\text{car} \ l)) \ (\text{lat?} \ (\text{cdr} \ l))) \\
\quad \quad \quad \quad (t^2 \ \text{nil})) \\
\text{) \}
\]

The application (lat? l) where l is (bacon and eggs) has the value t—true—because l is a lat

What is the value of (lat? l), where l is the argument (bacon and eggs)
How do you determine the answer $i$ for the application (lat? $l$)?

We did not expect you to know this one either. The answer is determined by answering the questions asked by lat?

Hint: Write down the function lat? and refer to it for the next group of questions.

What is the first question asked by (lat? $l$)? (null? $l$)

Note: (cond .) is the one that asks questions; (lambda ...) creates a function; and (define ...) gives it a name.

What is the meaning of the cond-line ((null? $l$) t),
where $l$ is (bacon and eggs) (null? $l$) asks if the argument $l$ is the null list. If it is, then the value of the application is true. If it is not, then we ask the next question. In this case, $l$ is not the null list, so we ask the next question.

What is the next question? (atom? (car $l$))

What is the meaning of the line ((atom? (car $l$)) (lat? (cdr $l$))),
where $l$ is (bacon and eggs) (atom? (car $l$)) asks if the first S expression of the list $l$ is an atom. If (car $l$) is an atom, then we want to know if the rest of $l$ is also composed only of atoms. If (car $l$) is not an atom, then we ask the next question. In this case, (car $l$) is an atom, so the value of the function is the value of (lat? (cdr $l$)).

What is the meaning of (lat? (cdr $l$)) (lat? (cdr $l$)) finds out if the rest of the list $l$ is composed only of atoms, by referring to the function, but now with a new argument.

Now, what is the argument $l$ for lat? Now the argument $l$ is (cdr $l$), which is (and eggs)
What is the meaning of the line
((null? l) t)
where
l is now (and eggs)

(null? l) asks if the argument l is the null list. If it is, then the value of the application is t. If it is not, then we ask the next question. In this case, l is not the null list, so we ask the next question.

What is the next question?

(atom? (car l))

What is the meaning of the line
((atom? (car l)) (lat? (cdr l)))
where
l is (and eggs)

(atom? (car l)) asks if (car l) is an atom. If it is an atom, then the value of the application is (lat? (cdr l)). If not, then we ask the next question. In this case, (car l) is an atom, so we want to find out if the rest of the list l is composed only of atoms.

What is the meaning of
(lat? (cdr l))

(lat? (cdr l)) finds out if the rest of l is composed only of atoms, by referring again to the function lat?, but this time, with the argument (cdr l), which is (eggs).

What is the next question?

(null? l)

What is the meaning of the line
((null? l) t)
where
l is now (eggs)

(null? l) asks if the argument l is the null list. If it is, the value of the application is t, namely true. If it is not, then move to the next question. In this case, l is not null, so we ask the next question.
What is the meaning of the line

((atom? (car l)) (lat? (cdr l)))

where

l is now (eggs)

(atom? (car l)) asks if (car l) is an atom. If it is, then the value of the application is (lat? (cdr l)). If (car l) is not an atom, then ask the next question. In this case, (car l) is an atom, so once again we look at (lat? (cdr l))

What is the meaning of (lat? (cdr l))

(lat? (cdr l)) finds out if the rest of the list l is composed only of atoms, by referring to the function lat?, with l becoming the value of (cdr l)

Now, what is the argument for lat? 

( )

What is the meaning of the line

((null? l) t)

where

l is now ( )

(null? l) asks if the argument l is the null list. If it is, then the value of the application is the value of t. If not, then we ask the next question. In this case, ( ) is the null list. Therefore, the value of the application (lat? l), where l is (bacon and eggs), is t—true.

Do you remember the question about (lat? l)

Probably not. The application (lat? l) has a value t if the list l is a list of atoms, where l is (bacon and eggs)

Can you describe what the function lat? does in your own words?

Here are our words

"lat? looks at each S-expression, in turn, and asks if each S-expression is an atom, until it runs out of S-expressions. If it runs out without encountering a list, the value is t. If it finds a list, the value is null—false."

To see how we could arrive at a value of "false," consider the next few questions
This is the function lat? again

\[
\text{(define lat?}
\begin{align*}
&\text{(lambda (l)} \\
&\text{\quad (cond) } \\
&\text{\quad \quad ((null? l) t} \\
&\text{\quad \quad \quad ((atom? (\text{car l})) (lat? (\text{cdr l}))} \\
&\text{\quad \quad \quad (t nil)))}
\end{align*}
\]

What is the value of (lat? l), where
l is now (bacon (and eggs))

nil,\(^1\)
since the list \(l\) contains an S expression that is a list

---

What is the first question?

\[(\text{null? } l)\]

What is the meaning of the line
\((\text{null? } l) t)\)
where
l is (bacon (and eggs))

\((\text{null? } l)\) asks if \(l\) is the null list. If it is, the value is \(t\). If \(l\) is not null, then move to the next question. In this case, it is not null, so we ask the next question.

What is the next question?

\[(\text{atom? (\text{car } l))}\]

What is the meaning of the line
\((\text{atom? (\text{car l)) (lat? (\text{cdr l))})}\)
where
l is (bacon (and eggs))

\((\text{atom? (\text{car l))})\) asks if \((\text{\text{car l})}\) is an atom. If it is, the value is \((\text{\text{lat? (\text{\text{cdr l})})})\). If it is not, we ask the next question. In this case, \((\text{\text{car l})}\) is an atom so we want to check if the rest of the list \(l\) is composed only of atoms.

What is the meaning of \((\text{\text{lat? (\text{\text{cdr l})})})\)?

\((\text{\text{lat? (\text{\text{\text{cdr l})})})}\) checks to see if the rest of the list \(l\) is composed only of atoms, by referring to \(\text{\text{lat?}}\) with \(l\) replaced by \((\text{\text{\text{cdr l})})\).

What is the meaning of the line
\((\text{null? } l) t)\)
where
l is now ((and eggs))

\((\text{null? } l)\) asks if \(l\) is the null list. If it is null, the value is \(t\). If it is not null, we ask the next question. In this case, \(l\) is not null, so move to the next question.
What is the meaning of the line 
\[ (((\text{atom?} \ (\text{car } l)) \ (\text{lat?} \ (\text{cdr } l))) \] 
where 
\[ l \text{ is now } ((\text{and eggs})) \]
\[(\text{atom?} \ (\text{car } l)) \text{ asks if } (\text{car } l) \text{ is an atom. If it is, then the value is } (\text{lat?} \ (\text{cdr } l)). \text{ If it is not, then we move to the next question. In this case, } (\text{car } l) \text{ is not an atom, so we ask the next question.} \]

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the next question?</td>
<td>t</td>
</tr>
<tr>
<td>What is the meaning of the question t?</td>
<td>t asks if t is true</td>
</tr>
<tr>
<td>Is t true?</td>
<td>Yes, because the question t is always true</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>Why is t the last question?</td>
<td>Because we do not need to ask any more questions</td>
</tr>
<tr>
<td>Why do we not need to ask any more questions?</td>
<td>Because a list can only be empty, or have an atom or a list in the first position</td>
</tr>
<tr>
<td>What is the meaning of the line (t nil)</td>
<td>t asks if t is true If t is true— as it always is— then the answer is nil— false</td>
</tr>
<tr>
<td>What is )))))</td>
<td>These are the closing or matching parentheses of (cond, (lambda, and (define, which appear at the beginning of a function definition. We sometimes call these &quot;aggravation parentheses,&quot; and they are always put at the end.</td>
</tr>
</tbody>
</table>
Can you describe how we determined the value \textit{nil} for \\
\texttt{(lat? l)} \\
where \\
\texttt{l} is \texttt{(bacon (and eggs))} \\

Here is one way to say it. \texttt{(lat? l)} looks at each item in its argument, 
to see if it is an atom. If it runs out of 
items before it finds a list, the value of 
\texttt{(lat? l)} is \texttt{t}. If it finds a list, as it did in 
the example \texttt{(bacon (and eggs))}, the value 
of \texttt{(lat? l)} is \texttt{nil}.

\begin{itemize}
\item Is \texttt{(or (null? l) (atom? s))} true or false, where \\
\texttt{l} is \texttt{( )}, and \\
\texttt{s} is \texttt{(d e f g)}
\item Is \texttt{(or (null? l1) (null? l2))} true or false, where \\
\texttt{l1} is \texttt{(a b c)}, and \\
\texttt{l2} is \texttt{( )}
\item Is \texttt{(or (null? l) (null? s))} true or false, where \\
\texttt{l} is \texttt{(a b c)}, and \\
\texttt{s} is \texttt{(atom)}
\end{itemize}

\begin{itemize}
\item True, 
because \texttt{(null? l)} is true where \texttt{l} is \texttt{ ( )}
\item True, 
because \texttt{(null? l2)} is true where \texttt{l2} is \texttt{( )}
\item False, 
because neither \texttt{(null? l)} is true where \texttt{l} 
is \texttt{(a b c)} nor \texttt{(null? s)} is true where \texttt{s} is 
\texttt{(atom)}
\end{itemize}

What does \texttt{(or \quad )} do? 
\texttt{(or \quad )} asks two questions, one at a time 
If the first one is true it stops and answers 
true. Otherwise \texttt{(or \quad )} asks the second 
question and answers with whatever the 
second question answers.

Is it true or false that \texttt{a} is a \textit{member of lat} 
where \\
\texttt{a} is \texttt{tea}, and \\
\texttt{lat} is \texttt{(coffee tea or milk)}

True, 
because one of the atoms of the \texttt{lat}, 
\texttt{(coffee tea or milk)} 
is the same as the atom \texttt{a}, namely \texttt{tea}
Is (member? a lat) true or false, where
   a is poached, and
lat is (fried eggs and scrambled eggs)

False,
since a is not one of the atoms of lat

This is the function member?

(define member?
  (lambda (a lat)
    (cond
      ((null? lat) nil)
      (t (or
        (eq? (car lat) a)
        (member? a (cdr lat)))))))

What is the value of (member? a lat), where
   a is meat, and
lat is (mashed potatoes and meat gravy)

How do we determine the value t for the above application?

The value is determined by asking the questions about (member? a lat).
  Hint: Write down the function member?
  and refer to it while you work on the next group of questions

What is the first question asked by
   (member? a lat)

(null? lat)
  This is also the first question asked by lat?

The First Commandment
Always ask null? as the first question in expressing any function.
What is the meaning of the line
((null? lat) nil)
where
lat is (mashed potatoes and meat gravy)

(null? lat) asks if lat is the null list. If it is, then the value is nil, since the atom meat was not found in lat. If not, then we ask the next question. In this case, it is not null, so we ask the next question.

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the next question?</td>
<td>t</td>
</tr>
<tr>
<td>Why is t the next question?</td>
<td>Because we do not need to ask any more questions</td>
</tr>
<tr>
<td>Is t really a question?</td>
<td>Yes, t is a question whose value is always true</td>
</tr>
</tbody>
</table>

What is the meaning of the line
(t (or
  (eq? (car lat) a)
  (member? a (cdr lat))))

Now that we know that lat is not null?, we have to find out whether the car of lat is the same atom as a, or whether a is somewhere in the rest of the lat. The question
  (or
    (eq? (car lat) a)
    (member? a (cdr lat)))

does this

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is</td>
<td>We will find out by looking at each question in turn</td>
</tr>
</tbody>
</table>
| (or
  (eq? (car lat) a)
  (member? a (cdr lat)))
true or false, where
a is meat, and
lat is (mashed potatoes and meat gravy) |
What is the second question for (or )
(member? a (cdr lat))
This refers to the function with the argument lat replaced by (cdr lat).

Now what are the arguments for member?
a is meat, and lat is now (cdr lat), specifically
(potatoes and meat gravy)

What is the next question?
(null? lat)
Remember The First Commandment

Is (null? lat) true or false, where
lat is (potatoes and meat gravy)
m1, namely false

What do we do now?
Ask the next question

What is the next question?
t

What is t?
t, namely true

What is the meaning of
(or
(eq? (car lat) a) (member? a (cdr lat)))
finds out if a is eq? to the car of lat or if a is a member of the cdr of lat by referring to the function.

Is a eq? to the car of lat
No, because a is meat and the car of lat is potatoes.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the next question?</td>
<td>(null? lat)</td>
</tr>
<tr>
<td>What do we do now?</td>
<td>Ask the next question, since (null? lat) is false</td>
</tr>
<tr>
<td>What is the next question?</td>
<td>t</td>
</tr>
<tr>
<td>What is the value of (or (eq? (car lat) a) (member? a (cdr lat)))</td>
<td>The value of (member? a (cdr lat))</td>
</tr>
<tr>
<td>Why?</td>
<td>Because (eq? (car lat) a) is false</td>
</tr>
<tr>
<td>What do we do now?</td>
<td>Recur—refer to the function with new arguments</td>
</tr>
<tr>
<td>What are the new arguments?</td>
<td>a is meat, and</td>
</tr>
<tr>
<td></td>
<td>lat is (meat gravy)</td>
</tr>
<tr>
<td>What is the next question?</td>
<td>(null? lat)</td>
</tr>
<tr>
<td>What do we do now?</td>
<td>Since (null? lat) is false, ask the next question</td>
</tr>
</tbody>
</table>
What is the value of
(or
  (eq? (car lat) a)
  (member? a (cdr lat)))

because (car lat), which is meat, and a, which is meat, are the same atom. Therefore, (or ..) answers with t.

What is the value of the application
(member? a lat)
where
a is meat, and
lat is (meat gravy)

t, because we have found that meat is a member of (meat gravy)

What is the value of the application
(member? a lat)
where
a is meat, and
lat is (and meat gravy)

t, because meat is also a member of the lat (and meat gravy)

What is the value of the application
(member? a lat)
where
a is meat, and
lat is (potatoes and meat gravy)

t, because meat is also a member of the lat (potatoes and meat gravy)

What is the value of the application
(member? a lat)
where
a is meat, and
lat is (mashed potatoes and meat gravy)

t, because meat is also a member of the lat (mashed potatoes and meat gravy).

Of course, you noticed that this is our original lat.
Just to make sure you have it right, let's quickly run through it again:

```
(define member? 
  (lambda (a lat)
    (cond
      ((null? lat) nil)
      (t (or
         (eq? (car lat) a)
         (member? a (cdr lat))))))))
```

What is the value of `(member? a lat)` where
  
  - `a` is meat, and
  - `lat` is (mashed potatoes and meat gravy)

---

```
(null? lat)
```

No  Move to the next line

```
t
```

Yes

```
(or
  (eq? (car lat) a)
  (member? a (cdr lat))))
```

Perhaps

```
(eq? (car lat) a)
```

No  Ask the next question

What next?

Recur with `a` and `(cdr lat)`, where
  
  - `a` is meat and
  - `(cdr lat)` is (potatoes and meat gravy)
(null? lat)

No Move to the next line

(t)

Yes, but (eq? (car lat) a) is false
Recur with a and (cdr lat), where
a is meat, and
(cdr lat) is (meat gravy)

(nul? lat)

No Move to the next line

(eq? (car lat) a)

Yes, the value is t

(or
  (eq? (car lat) a)
  (member? a (cdr lat)))

What is the value of (member? a lat), where
a is meat, and
lat is (meat gravy)

What is the value of (member? a lat), where
a is meat, and
lat is (and meat gravy)

What is the value of (member? a lat), where
a is meat, and
lat is (potatoes and meat gravy)
What is the value of `(member? a lat)`, where
- `a` is meat, and
- `lat` is (mashed potatoes and meat gravy)

What is the value of `(member? a lat)`, where
- `a` is liver, and
- `lat` is (bagels and lox)

Let's work out why it is `nil`. What's the first question `member?` asks?

<table>
<thead>
<tr>
<th><code>(null? lat)</code></th>
<th>No</th>
<th>Move to the next line</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t</code></td>
<td>Yes, but <code>(eq? (car lat) a)</code> is false. Recur with <code>a</code> and <code>(cdr lat)</code>, where <code>a</code> is liver, and <code>(cdr lat)</code> is (and lox)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>(null? lat)</code></th>
<th>No</th>
<th>Move to the next line</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t</code></td>
<td>Yes, but <code>(eq? (car lat) a)</code> is false Recur with <code>a</code> and <code>(cdr lat)</code>, where <code>a</code> is liver, and <code>(cdr lat)</code> is (lox)</td>
<td></td>
</tr>
</tbody>
</table>
(null? lat)  Yes

What is the value of (member? a lat), where
  a is liver, and
  lat is ()
  nil

What is the value of
  (or
    (eq? (car lat) a)
    (member? a (cdr lat)))
where
  a is liver, and
  lat is (lox)
  nil

What is the value of (member? a lat), where
  a is liver, and
  lat is (lox)
  nil

What is the value of
  (or
    (eq? (car lat) a)
    (member? a (cdr lat)))
where
  a is liver, and
  lat is (and lox)
  nil

What is the value of (member? a lat) where
  a is liver, and
  lat is (and lox)
  nil
What is the value of \( \text{nil} \)
\[
(\text{or} \\
\quad (\text{eq?} \ (\text{car} \ lat) \ a)) \\
\quad (\text{member?} \ a \ (\text{cdr} \ lat)))
\]
where
- \( a \) is liver, and
- \( lat \) is \((\text{bagels} \ \text{and} \ \text{lox})\)

What is the value of \( \text{nil} \), where
- \( a \) is liver, and
- \( lat \) is \((\text{bagels} \ \text{and} \ \text{lox})\)

Do you believe all this? Then you may rest!
Exercises

For these exercises,

\[ l1 \text{ is (german chocolate cake)} \]
\[ l2 \text{ is (poppy seed cake)} \]
\[ l3 \text{ is ((linzer) (torte) ( ))} \]
\[ l4 \text{ is ((bleu cheese) (and) (red) (wine))} \]
\[ l5 \text{ is (( ) ( ))} \]
\[ a1 \text{ is coffee} \]
\[ a2 \text{ is seed} \]
\[ a3 \text{ is poppy} \]

2.1 What are the values of \( \text{(lat? l1)}, \text{(lat? l2)}, \text{and (lat? l3)}? \)

2.2 For each case in Exercise 2.1 step through the application as we did in this chapter.

2.3 What is the value of \( \text{(member? a1 l1)}, \text{and (member? a2 l2)}? \) Step through the application for each case.

2.4 Most Lisp dialects have an \( \text{(if )} \) form. In general an \( \text{(if )} \) form looks like this:
\[ \text{(if aexp bexp cexp)} \]

When \( aexp \) is true, \( \text{(if aexp bexp cexp)} \) is \( bexp \); when it is false, \( \text{(if aexp bexp cexp)} \) is \( cexp \) example,

\[ \text{(cond} \]
\[ \text{((null? l) nil)} \]
\[ \text{(t (or} \]
\[ \text{ (eq? (car l) a)} \]
\[ \text{ (member? a (cdr l))))))} \]

\text{in member? can be replaced by:}
(if (null? l)
   nil
   (or
    (eq? (car l) a)
    (member? a (cdr l))))

Rewrite all the functions in the chapter using (if ) instead of (cond )

2.5 Write the function nonlat? which determines whether a list is the empty list or does not contain atomic S-expressions

Example: (nonlat? l1) is false,
         (nonlat? l2) is false,
         (nonlat? l3) is false,
         (nonlat? l4) is true

2.6 Write a function member-cake? which determines whether a list contains the atom cake

Example: (member-cake? l1) is true,
         (member-cake? l2) is true,
         (member-cake? l5) is false.

2.7 Consider the following new definition of member?

(define member2?
  (lambda (a lat)
    (cond
     ((null? lat) nil)
     (t (or
      (member2? a (cadr lat))
      (eq? a (car lat)))))))

Do (member2? a l) and (member? a l) give the same answer when we use the same arguments? Consider the examples a1 and l1, a1 and l2, and a2 and l2

2.8 Step through the applications (member? a3 l2) and (member2? a3 l2). Compare the steps of the two applications

2.9 What happens when you step through (member? a2 l3)? Fix this problem by having member? ignore lists

2.10 The function member? tells whether some atom appears at least once in a list. Write a function member-twice? which tells whether some atom appears at least twice in a list
Cons The Magnificent
<table>
<thead>
<tr>
<th>What is (rember a lat) where a is mint, and lat is (lamb chops and mint jelly)</th>
<th>(lamb chops and jelly) &quot;Rember&quot; stands for remove a member</th>
</tr>
</thead>
<tbody>
<tr>
<td>(rember a lat) where a is mint, and lat is (lamb chops and flavored mint jelly)</td>
<td>(lamb chops and flavored mint jelly)</td>
</tr>
<tr>
<td>(rember a lat), where a is toast, and lat is (bacon lettuce and tomato)</td>
<td>(bacon lettuce and tomato)</td>
</tr>
<tr>
<td>(rember a lat), where a is cup, and lat is (coffee cup tea cup and hick cup)</td>
<td>(coffee tea cup and hick cup)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What does (rember a lat) do?</th>
<th>It takes an atom and a lat as its arguments, and makes a new lat with the first occurrence of the atom in the old lat removed</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>What steps will we use to do this?</th>
<th>First we will test (null? lat) — The First Commandment</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>And if (null? lat) is true?</th>
<th>Return ()</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>What do we know if (null? lat) is not true?</th>
<th>We know that there must be at least one atom in the lat</th>
</tr>
</thead>
</table>
How do we ask questions?

By using

\[
\text{(cond} \\
\text{)}
\]

How do we ask if \( a \) is the same as \((\text{car } \text{lat})\)?

(eq? (car lat) a)

What would be the value of \((\text{rember } a \text{ lat})\) if \( a \) were the same as \((\text{car } \text{lat})\)?

(cdr lat)

What do we do if \( a \) is not the same as \((\text{car } \text{lat})\)?

We will want to keep \((\text{car } \text{lat})\), but also find out if \( a \) is somewhere in the rest of the \text{lat}

How do we replace the first occurrence of \( a \) in the rest of \text{lat}?

(rember a (cdr lat))

Is there any other question we should ask?

No

Now, let's write down what we have so far

\[
\text{(define rember} \\
\text{\ (lambda} (a \text{ lat)} \\
\text{\ (cond} \\
\text{\ \ \ \ ((null? \text{ lat) (quote ( ))))} \\
\text{\ \ \ \ (t (cond} \\
\text{\ \ \ \ \ \ \ \ ((eq? (car \text{ lat) a) (cdr \text{ lat}))} \\
\text{\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ (t (rember a (cdr \text{ lat)))})\])))})
\]

What is the value of \((\text{rember } a \text{ lat})\) where \( a \) is bacon, and \( \text{lat} \) is \((\text{bacon lettuce and tomato})\)?

\[
\text{Hint: Write down the function rember and its arguments, and refer to them as you go through the next sequence of questions}
\]
<table>
<thead>
<tr>
<th>What next?</th>
<th>Ask the next question</th>
</tr>
</thead>
<tbody>
<tr>
<td>(eq? (car lat) a)</td>
<td>Yes, so the value is (cdr lat). In this case, it is the list (lettuce and tomato)</td>
</tr>
<tr>
<td>Is this the correct value?</td>
<td>Yes, because the above list is the original list without the atom bacon.</td>
</tr>
<tr>
<td>But did we really use a good example?</td>
<td>Who knows? But the proof of the pudding is in the eating, so let's try another example.</td>
</tr>
<tr>
<td>What does rember do?</td>
<td>It takes an atom and a lat as its arguments, and makes a new lat with the first occurrence of the atom in the old lat removed.</td>
</tr>
<tr>
<td>What will we do?</td>
<td>We will compare each atom of the lat with the atom a, and if the comparison fails we will build a list which begins with the atom we just compared</td>
</tr>
<tr>
<td>What is the value of (rember a lat), where a is and, and lat is (bacon lettuce tomato)</td>
<td>(bacon lettuce tomato)</td>
</tr>
</tbody>
</table>
Let us see if our function rember works
What is the first question asked by rember

What do we do now?

$$(\text{null? } lat)$$

Move to the next line, and ask the next question.

t

$$(eq\ (\text{car lat}) a)$$

No, so move to the next line

What is the meaning of
$$(t\ (\text{rember a (cdr lat)}))$$

t asks if $t$ is true—as it always is—and the rest of the line says to recur with $a$ and
(cdr lat), where
$a$ is and, and
(cdr lat) is (lettuce and tomato)

$$(\text{null? } lat)$$

No, so move to the next line

t

$$(eq\ (\text{car lat}) a)$$

No, so move to the next line

What is the meaning of
$$(t\ (\text{rember a (cdr lat)}))$$

Recur, where
$a$ is and, and
(cdr lat) is (and tomato)
What is the value of the application
(rember a lat)

(cdr lat), that is (tomato)

Is this correct?

No, since (tomato) is not the list
(bacon lettuce and tomato)
with only a, namely and, removed

What did we do wrong?

We dropped and, but we also lost all the
atoms preceding and

How can we keep from losing the atoms
bacon and lettuce

We use Cons The Magnificent Remember
cons, from Chapter 1?

---

The Second Commandment

Use cons to build lists.

---

Let's see what happens when we use cons

(define rember
  (lambda (a lat)
    (cond
      (((null? lat) (quote ( )))
       (t (cond
          ((eq? (car lat) a) (cdr lat))
          (t (cons (car lat)
                  (rember
                   a (cdr lat))))))))))

(bacon lettuce tomato)

Make a copy of this function with cons
and the arguments a and lat so you can
refer to it for the following questions

What is the value of (rember a lat), where
a is and, and
lat is (bacon lettuce and tomato)
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(eq? (car lat) a)</code></td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>What is the meaning of (cons <code>\(\text{\text{n}}\)</code>) (car lat) (rember a (cdr lat)))</td>
<td>cons (car lat)—that is, bacon—onto the value of (rember a (cdr lat))</td>
</tr>
<tr>
<td>where</td>
<td>But since we don't know the value of (rember a (cdr lat)) yet, we will have to find it before we can cons (car lat) onto it.</td>
</tr>
<tr>
<td><code>a</code> is and, and</td>
<td></td>
</tr>
<tr>
<td>lat is (bacon lettuce and tomato)</td>
<td></td>
</tr>
<tr>
<td>What is the meaning of (rember a (cdr lat))</td>
<td>This refers to the function, with lat replaced by (cdr lat), that is, (lettuce and tomato)</td>
</tr>
<tr>
<td>(null? lat)</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>t</td>
<td>Yes, ask the next question</td>
</tr>
<tr>
<td><code>(eq? (car lat) a)</code></td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>What is the meaning of (cons <code>\(\text{\text{n}}\)</code>) (car lat) (rember a (cdr lat)))</td>
<td>It means cons (car lat), namely lettuce, onto the value of (rember a (cdr lat)). But since we don't know the value, we must first find that value before consing (car lat) onto it.</td>
</tr>
</tbody>
</table>
What is the meaning of (rember a (cdr lat))

This refers to the function with lat replaced by (cdr lat), that is, (and tomato)

(null? lat)

No, so ask the next question

t

Still

(eq? (car lat) a)

Yes

What is the value of the line

((eq? (car lat) a) (cdr lat))

(cdr lat), that is, (tomato)

Are we finished?

Certainly not! We know what (rember a lat) is when lat is (and tomato), but we don’t yet know what it is when lat is (lettuce and tomato) or (bacon lettuce and tomato)

We now have a value for

(rember a (cdr lat)), where

a is and,

and

(cdr lat) is (and tomato)

This value is (tomato) What next?

Recall that we wanted to cons lettuce onto the value of (rember a (cdr lat)), where a was and and (cdr lat) was (and tomato).

Now that we have this value, which is (tomato), we can cons lettuce onto this value

What is the result when we cons lettuce onto (tomato)

(lettuce tomato)

What does (lettuce tomato) represent?

It represents the value of

(cons

(car lat)

(rember a (cdr lat))),

when

lat was (lettuce and tomato), and

(rember a (cdr lat)) was (tomato)
We now have a value for \((\text{rember} \ a \ (\text{cdr} \ \text{lat}))\) when
\[ a \] is and, and
\[ (\text{cdr} \ \text{lat}) \] is (lettuce and tomato)
This value is (lettuce tomato). This is not the final value, so what must we do again?

Recall that, at one time, we wanted to cons bacon onto the value of \((\text{rember} \ a \ (\text{cdr} \ \text{lat}))\), when
\[ a \] was and, and
\[ (\text{cdr} \ \text{lat}) \] was (lettuce and tomato).
Now that we have this value, which is (lettuce tomato), we can cons bacon onto this value.

What is the result when we cons bacon onto (lettuce tomato)?

(bacon lettuce tomato)

What does (bacon lettuce tomato) represent?†

It represents the value of
\[ \text{cons} \]
\[ (\text{car} \ \text{lat}) \]
\[ (\text{rember} \ a \ (\text{cdr} \ \text{lat})) \]
when
\[ \text{lat} \] was (bacon lettuce and tomato), and
\[ (\text{rember} \ a \ (\text{cdr} \ \text{lat})) \] was (lettuce tomato)

Are we finished yet?

Yes

Can you put in your own words how we determined the final value (bacon lettuce tomato)

In our words
"Rember checked each atom of the lat, one at a time, to see if it was the same as the atom and If the car was not the same as the atom, we saved it to be consed to the final value later. When rember found the atom and, it dropped it, and consed the previous atoms onto the rest of the lat, in reverse order."
Can you rewrite rember so that it reflects the above description?

Yes, we can simplify it

\[
\text{(define rember}
\text{(lambda (a lat)}
\text{(cond}
\text{((null? lat) (quote ())})
\text{((eq? (car lat) a) (cdr lat))}
\text{(t (cons (car lat)
\text{(rember a (cdr lat))))}))}
\]

Do you think this is simpler?

Functions like rember can always be simplified in this manner

So why don’t we simplify yet?

Because then a function’s structure does not coincide with its data’s structure.

Let’s see if the new rember is the same as the old one. What is the value of the application

\[\text{(rember a lat),}\]

where

\[a\] is and, and

\[lat\] is \(\text{(bacon lettuce and tomato)}\)

(bacon lettuce tomato).

Hint: Write down the function rember and its arguments and refer to them as you go through the next sequence of questions

\[\text{(null? lat)}\]

No

\[\text{(eq? (car lat) a)}\]

No

\[t\]

Yes, so the value is

\[\text{(cons}
\text{(car lat)}
\text{(rember a (cdr lat))))}\]
What is the meaning of
\[(\text{cons}
\begin{array}{l}
(\text{car} \; \text{lat}) \\
(\text{rember} \; a \; (\text{cdr} \; \text{lat})))
\end{array}\]
This says to refer to the function rember, but with the argument \(\text{lat}\) replaced by \((\text{cdr} \; \text{lat})\), and that after we arrive at a value for \((\text{rember} \; a \; (\text{cdr} \; \text{lat}))\) we will cons \((\text{car} \; \text{lat})\), namely bacon, onto it

\((\text{null?} \; \text{lat})\)
No

\((\text{eq?} \; (\text{car} \; \text{lat}) \; a)\)
No

\(t\)
Yes, so the value is
\[\begin{array}{l}
(\text{cons}
\begin{array}{l}
(\text{car} \; \text{lat}) \\
(\text{rember} \; a \; (\text{cdr} \; \text{lat})))
\end{array}
\end{array}\]

What is the meaning of
\[(\text{cons}
\begin{array}{l}
(\text{car} \; \text{lat}) \\
(\text{rember} \; a \; (\text{cdr} \; \text{lat})))
\end{array}\]
This says we recur using the function rember, with the argument \(\text{lat}\) replaced by \((\text{cdr} \; \text{lat})\), and that after we arrive at a value for \((\text{rember} \; a \; (\text{cdr} \; \text{lat}))\), we will cons \((\text{car} \; \text{lat})\), namely lettuce, onto it

\((\text{null?} \; \text{lat})\)
No

\((\text{eq?} \; (\text{car} \; \text{lat}) \; a)\)
Yes

What is the value of the line
\[((\text{eq?} \; (\text{car} \; \text{lat}) \; a) \; (\text{cdr} \; \text{lat}))\]
It is \((\text{cdr} \; \text{lat})\), that is, \((\text{tomato})\)
Now what?

Now cons (car lat), that is, bacon, onto (lettuce tomato).

Now that we have completed rember, try this example

(rember a lat),
where

$a$ is sauce, and

$lat$ is (soy sauce and tomato sauce)

(rember a lat) is (soy and tomato sauce)

What is (firsts $l$), where

$l$ is ((apple peach pumpkin)

(plum pear cherry)

(grape raisin pea)

(bean carrot eggplant))

(apple plum grape bean)

What is (firsts $l$), where

$l$ is ((a b) (c d) (e f))

(a c e)

What is (firsts $l$), where

$l$ is ( )

( )

What is (firsts $l$), where

$l$ is ((five plums) (four)

(eleven green oranges))

(five four eleven)

In your own words, what does (firsts $l$) do?

We tried the following

"Firsts takes one argument, a list, which must either be a null list, or contain only non-null lists. It builds another list composed of the first $S$ expression of each internal list."
See if you can write the function firsts

\textit{Remember the Commandments!}

Believe it or not, you can probably write the following:

\[
\begin{align*}
&\text{(define firsts} \\
&\quad \text{(lambda} \ (l) \\
&\quad \quad \text{(cond}) \\
&\quad \quad \quad \text{(null?} \ l \ \text{____)}) \\
&\quad \quad \quad \text{(t} \ (\text{cons} \ \text{____} \ \text{(firsts} \ \text{(cdr} \ l))))}))
\end{align*}
\]

Why (define firsts (lambda (l))

Because we always state the function name, (lambda, and the argument(s) of the function

Why (cond

Because we need to ask questions about the actual arguments.

Why ((null? l) _____ )

The First Commandment

Why (t

Because we only have two questions to ask about the list l: either it is the null list, or it contains at least one non-null list

Why (t

See above And because the last question is always t

Why (cons

Because we are building a list

---The Second Commandment

Why (firsts (cdr l))

Because we can only look at one S-expression at a time To do this, we must recur.
Because these are the matching parentheses for `(cond, (lambda, and (define, and they always appear at the end of a function definition

Keeping in mind the definition of `(firsts l),
what is a *typical element* of the value of
(firsts l), where
  \( l = ((a\ b)\ (c\ d)\ (e\ f)) \)

What is another typical element?  
\( c \) or \( e \)

Consider the function `seconds`. What would be a typical element of the value of
(`seconds l), where
  \( l = ((a\ b)\ (c\ d)\ (e\ f)) \)

How do we *describe* a typical element for
(firsts l)  
As the car of (car l), that is, (car (car l))
See Chapter 1

When we find a typical element of (firsts l),
what do we do with it?  
We cons it onto the recursion, that is,
(firsts (cdr l)).

---

**The Third Commandment**

When building a list, describe the first typical element, and then cons it onto the natural recursion.
With The Third Commandment, we can now fill in more of the function firsts. What does the last line look like now?

\[(t \ (\text{cons} \ (\text{car} \ (\text{car} \ l)) \ (\text{firsts} \ (\text{cdr} \ l))))\]

	typical element

natural recursion

What does \((\text{firsts} \ l)\) do

\[
(\text{define} \ \text{firsts} \\
(\lambda \ l) \\
(\text{cond} \\
\quad ((\text{null?} \ l) \ \text{---}) \\
\quad \quad (t \ (\text{cons} \ (\text{car} \ (\text{car} \ l)) \\
\quad \quad \quad (\text{firsts} \ (\text{cdr} \ l))))))))
\]

where

\(l\) is \(((a \ b) \ (c \ d) \ (e \ f))\)

\((\text{null?} \ l)\), where

\(l\) is \(((a \ b) \ (c \ d) \ (e \ f))\)

Nothing yet. We are still missing one important ingredient in our recipe. The first line \((\text{null?} \ l) \ \\
\quad \text{---}\) needs a value for the case where \(l\) is the null list. We can, however, proceed without it for now.

No, so move to the next line

\((\text{null?} \ l)\), where

\(l\) is \(((c \ d) \ (e \ f))\)

What is the meaning of

\(\text{cons} \\
\quad (\text{car} \ (\text{car} \ l)) \\
\quad (\text{firsts} \ (\text{cdr} \ l)))\)

It saves \((\text{car} \ (\text{car} \ l))\) to \(\text{cons}\) onto \((\text{firsts} \ (\text{cdr} \ l))\). To find \((\text{firsts} \ (\text{cdr} \ l))\), we refer to the function with the new argument \((\text{cdr} \ l)\)

No, so move to the next line

What is the meaning of

\(\text{cons} \\
\quad (\text{car} \ (\text{car} \ l)) \\
\quad (\text{firsts} \ (\text{cdr} \ l)))\)

Save \((\text{car} \ (\text{car} \ l))\), and recur with \((\text{firsts} \ (\text{cdr} \ l))\).
(null? l), where l is ((e f))

No, so move to the next line

What is the meaning of (cons (car (car l)) (firsts (cdr l)))

Save (car (car l)), and recur with (firsts (cdr l))

(nul? l)

Yes

Now what is the value of the line ((null? l) ______ )

There is no value, something is missing

What do we need to cons atoms onto?

A list

Remember The Law of Cons?

What value can we give when (null? l) is true, for the purpose of consing?

Since the final value must be a list, we cannot use t or ml. Let's try (quote ())
With () as a value, we now have three cons steps to go back and pick up (a c e)

I  We need to:
   1. cons e onto ( )
   2. cons c onto the value of 1
   3. cons a onto the value of 2

or, alternatively,

II  We need to
   1. cons a onto the value of 2
   2. cons c onto the value of 3
   3. cons e onto ( )

or, alternatively,

III  We need to
    cons a onto
        the cons of c onto
        the cons of e onto
        ( )

In any case, what is the final value of (firsts l)

With which of the three alternatives do you feel most comfortable?

Correct! Now you use that one

What is (insert new old lat)
where
   new is topping,
   old is fudge, and
   lat is (ice cream with fudge for dessert)

(insert new old lat), where
   new is jalapeño,
   old is and, and
   lat is (tacos tamales and salsa)
<table>
<thead>
<tr>
<th>(insertr new old lat), where new is e, old is d, and lat is (a b c d f g d h)</th>
<th>(a b c d e f g d h)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In your own words, what does (insertr new old lat) do?</td>
<td>In our words “It takes three arguments: the atoms new and old, and a lat. Insertr builds a lat with new inserted to the right of the first occurrence of old”</td>
</tr>
<tr>
<td>See if you can write the first three lines of the function insertr.</td>
<td>(define insertr</td>
</tr>
<tr>
<td></td>
<td>(lambda (new old lat)</td>
</tr>
<tr>
<td></td>
<td>(cond</td>
</tr>
<tr>
<td>Which argument will change when we recur with insertr?</td>
<td>lat, because we can only look at one of its atoms at a time</td>
</tr>
<tr>
<td>How many questions can we ask about lat?</td>
<td>Two</td>
</tr>
<tr>
<td></td>
<td>A lat is either the null list or a non-null list of atoms</td>
</tr>
<tr>
<td>Which questions will we ask?</td>
<td>First, we will ask (null? lat) Second, we will ask t, because t is always the last question</td>
</tr>
<tr>
<td>What do we know if (null? lat) is not true?</td>
<td>We know that there is at least one element in lat</td>
</tr>
<tr>
<td>Which questions will we ask about the first element?</td>
<td>First, we will ask (eq? (car lat) old). Then we ask t, because there are no other interesting cases</td>
</tr>
</tbody>
</table>
Now see if you can write the whole function `insertr`

```
(define insertr
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
        ((eq? (car lat) old) (cdr lat))
        (t (cons (car lat)
           (insertr
             new old (cdr lat))))))))
```

Here is our first attempt

```
(define insertr
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
        ((eq? (car lat) old) (cdr lat))
        (t (cons (car lat)
           (insertr
             new old (cdr lat))))))))
```

What is the value of the `insertr` we just wrote, where
`new` is topping, `old` is fudge, and `lat` is (ice cream with fudge for dessert)

(ice cream with for dessert)

Notice that so far, this is the same as remember; but for `insertr`, what do we do when `(eq? (car lat) old)` is true?

When `(car lat)` is the same as `old`, we want to insert `new` to the right

How is this done?

Let's try consing `new` onto `(cdr lat)`

Now we have

```
(define insertr
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
        ((eq? (car lat) old)
         (cons new (cdr lat)))
        (t (cons (car lat)
           (insertr
             new old (cdr lat))))))))
```

Yes
So what is \((\text{insertR } new \ old \ lat)\) now, where \(new\) is topping, \(old\) is fudge, and \(lat\) is (ice cream with fudge for dessert) 

\(\text{(ice cream with topping for dessert)}\)

---

**Is this the list we wanted?**

No, we have only replaced fudge with topping

---

**What still needs to be done?**

Somehow we need to include the atom which is the same as \(old\) before the atom \(new\)

---

**How can we include \(old\) before \(new\)?**

Try consing \(old\) onto \((\text{cons new (cdr lat)})\)

---

Now you should be able to write the rest of the function \(\text{insertR}\). Do it

\[
\begin{align*}
\text{(define insertR} \\
\text{(lambda (new old lat) } \\
\text{(cond} \\
\text{(null? lat) (quote ( ))} \\
\text{(t (cond} \\
\text{(eq? (car lat) old) } \\
\text{(cons old} \\
\text{(cons new (cdr lat)))}) \\
\text{(t (cons (car lat} \\
\text{(insertR} \\
\text{new old (cdr lat)))))))))))
\end{align*}
\]

When \(new\) is topping, \(old\) is fudge, and \(lat\) is (ice cream with fudge for dessert), the value of the application, \((\text{insertR } new \ old \ lat)\), is (ice cream with fudge topping for dessert)

If you got this right, have one
Now try insertL

Hint: insertL inserts the atom new to the left of the first occurrence of the atom old in lat.

This much is easy, right?

(define insertL
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
        ((eq? (car lat) old)
          (cons new
            (cons old (cdr lat))))
        (t (cons (car lat)
          (insertL
           new old (cdr lat)))))))))

Did you think of a different way to do it?

For example,

((eq? (car lat) old)
  (cons new (cons old (cdr lat))))

could have been

((eq? (car lat) old)
  (cons new lat)),
since (cons old (cdr lat)) where old is eq? to (car lat) is the same as lat

Now try subst

Hint: (subst new old lat) replaces the first occurrence of old in the lat with the atom new. For example, where
new is topping,
old is fudge, and
lat is (ice cream with fudge for dessert),
the value is
(ice cream with topping for dessert)
Now you have the idea

Obviously,

(define subst
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
        ((eq? (car lat) old)
          (cons new (cdr lat)))
        (t (cons (car lat)
          (subst
           new old (cdr lat)))))))))

This is the same as our second attempt at insertR
Go cons a piece of cake onto your mouth

Now try subst2

Hint:

 subst2 new o1 o2 lat
 replaces either the first occurrence of o1
 or the first occurrence of o2 by new  For
 example, where
 new is vanilla,
 o1 is chocolate,
 o2 is banana, and
 lat is (banana ice cream
 with chocolate topping)
 the value is
 (vanilla ice cream
 with chocolate topping)

(define subst2
 (lambda (new o1 o2 lat)
 (cond
 ((null? lat) (quote ( )))
 (t (cond
 ((eq? (car lat) o1)
 (cons new (cdr lat)))))
 ((eq? (car lat) o2)
 (cons new (cdr lat))))
 (t (cons (car lat)
 (subst2 new
 o1 o2 (cdr lat)))))
))

Did you think of a better way?

Replace the two eq? lines about the (car lat) by

((or (eq? (car lat) o1) (eq? (car lat) o2))
 (cons new (cdr lat)))

If you got the last function, go and repeat the cake consing
For these exercises,

\[ l1 \text{ is } ((\text{paella spanish}) \ (\text{wine red}) \ (\text{and beans})) \]
\[ l2 \text{ is } (\ ) \]
\[ l3 \text{ is } (\text{cincinnati chili}) \]
\[ l4 \text{ is } (\text{texas hot chili}) \]
\[ l5 \text{ is } (\text{soy sauce and tomato sauce}) \]
\[ l6 \text{ is } ((\text{spanish}) \ (\ ) \ (\text{paella})) \]
\[ l7 \text{ is } ((\text{and hot}) \ (\text{but dogs})) \]
\[ a1 \text{ is } \text{chili} \]
\[ a2 \text{ is } \text{hot} \]
\[ a3 \text{ is } \text{spicy} \]
\[ a4 \text{ is } \text{sauce} \]
\[ a5 \text{ is } \text{soy} \]

3.1 Write the function seconds which takes a list of lats and makes a new lat consisting of the second atom from each lat in the list.

Example: (seconds \( l1 \)) is (spanish red beans)

(seconds \( l2 \)) is ( )

(seconds \( l7 \)) is (hot dogs)

3.2 Write the function dupla of \( a \) and \( l \) which makes a new lat containing as many \( a \)'s as there are elements in \( l \).

Example: (dupla a2 l4) is (hot hot hot)

(dupla a2 l2) is ( )

(dupla a1 l5) is (chili chili chili chili chili)

3.3 Write the function double of \( a \) and \( l \) which is a converse to rember. The function doubles the first \( a \) in \( l \) instead of removing it.

Example: (double a2 l2) is ( )

(double a1 l3) is (cincinnati chili chili)

(double a2 l4) is (texas hot hot chili)
3.4 Write the function subst sauce of a and l which substitutes a for the first atom sauce in l
Example: (subst-sauce a1 l4) is (texas hot chili)
            (subst-sauce a1 l5) is (soy chili and tomato sauce)
            (subst-sauce a4 l2) is ( )

3.5 Write the function subst3 of new, a1, a2, a3 and lat which—like subst2—replaces the first
occurrence of either a1, a2, or a3 in lat by new
Example (subst3 a5 a1 a2 a4 l5) is (soy soy and tomato sauce)
           (subst3 a4 a1 a2 a3 l4) is (texas sauce chili)
           (subst3 a3 a1 a2 a5 l2) is ( )

3.6 Write the function substN of new, slat and lat which replaces the first atom in lat that
also occurs in slat by the atom new
Example: (substN a2 l3 l4) is (texas hot hot)
          (substN a4 l3 l5) is (soy sauce and tomato sauce)
          (substN a4 l3 l2) is ( )

3.7 Step through the application (rember a4 l5). Also step through (insertr a5 a2 l5) for the
“bad” definitions of insertr.

3.8 Determine the typical elements and the natural recursions of the functions that you have
written so far

3.9 Write the function rember2 of a and l which removes the second occurrence of a m l
Example: (rember2 a1 l3) is (cincinnati chili)
          (rember2 a4 l5) is (soy sauce and tomato)
          (rember2 a4 l2) is ( )

Hint: Use the function rember

3.10 Consider the functions insertr, insertl, and subst. They are all very similar. Write the
functions next to each other and draw boxes around the parts that they have in common. Can
you see what rember has in common with these functions?
Numbers Games
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is 14 an atom?</td>
<td>Yes because all numbers are atoms</td>
</tr>
<tr>
<td>Is ((\text{atom? } n)) true or false where (n) is 14</td>
<td>True, because 14 is an atom</td>
</tr>
<tr>
<td>Is (-3) a number?</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>However, we do not consider negative numbers in this book</td>
</tr>
<tr>
<td>Is (31415) a number?</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>However, we consider only whole numbers in this book</td>
</tr>
<tr>
<td>Are (3) and (31415) numbers?</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>However, the only numbers we use are the nonnegative integers (i.e., 0, 1, 2, 3, 4, \ldots)</td>
</tr>
<tr>
<td>What is ((\text{add}^1 n)) where (n) is 67</td>
<td>68</td>
</tr>
<tr>
<td>What is ((\text{sub}^1 n)), where (n) is 5</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^{1}\) L: t^+  
S: See Preface page xix
Is \((\text{zero? } n)\) true or false, where 
\(n\) is 0

True

Is \((\text{zero? } n)\) true or false, where 
\(n\) is 1492

False

What is \((+ n m)\), where 
\(n\) is 46 and 
\(m\) is 12

58

Try to write the function +

Hint: It uses zero?, add1, and sub1

\[
\text{(define +}
\text{  (lambda (n m)}
\text{    (cond}
\text{      ((zero? m) n)}
\text{      (t (add1 (+ n (sub1 m)))))}}
\]

Wasn't that easy?

But didn't we just violate The First Commandment?

Yes! However, we can treat zero? like null? since zero? asks if a number is empty and null? asks if a list is empty

If zero? is like null?, is add1 like cons?

Yes! cons builds lists and add1 builds numbers.
What is \((- n m)\), where

- \(n\) is 17, and
- \(m\) is 9

No answer ¹

¹ No negative numbers

Try to write the function –

Hint: Use sub1.

How about this

\[
(\text{define -}
  
  (\lambda (n m)
    (\text{cond}
      
      ((\text{zero? } m) n)
      
      (t (\text{sub1 } (- n (\text{sub1 } m))))))
)
\]

Can you describe how \((- n m)\) does what it does?

It takes two numbers as arguments, and reduces the second until it hits zero. It subtracts one from the first number as many times as it did to cause the second one to reach zero.

Is this a vec?

\((2 \ 1 \ 1 \ 3 \ 7 \ 9 \ 4 \ 7 \ 6)\)

Yes, because it is a list of numbers

Is this a vec?

\((1 \ 2 \ 8 \ x \ 4 \ 3)\)

No, it is just a list of atoms
Is this a vec?
(3 7 4 13 9)  
No, because it is not a list of numbers (7 4) is not a number

Is this a vec?
( )  
Yes, it is a list of zero numbers. This special case is the empty vec

What is \(\text{addvec vec}\), where vec is (3 5 2 8)
18

What is \(\text{addvec vec}\), where vec is (15 6 7 12 3)
43

What does addvec do?
It builds a number by totalling all the numbers in its argument.

What is the natural way to build numbers from a list, just as cons is the natural way to build lists?
By using + in place of cons

When building lists with cons, the value of the terminal condition is ( ). What should be the value of the terminal condition when building numbers with +
0

What is the natural terminal condition for a list?
(null? l)
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>When you want to build a number from a list of numbers, what should the terminal condition line look like?</td>
<td><code>((null? vec) 0)</code>, just as <code>((null? l) (quote ( )))</code> is often the terminal condition line for lists</td>
</tr>
<tr>
<td>What is the terminal condition line of addvec?</td>
<td><code>((null? vec) 0)</code></td>
</tr>
<tr>
<td>How is a lat defined?</td>
<td>It is either an empty lat, or it contains an atom and a rest, where rest is again a lat</td>
</tr>
<tr>
<td>How is a vec defined?</td>
<td>It is either an empty vec, or it contains a number and a rest, where rest is again a vec</td>
</tr>
<tr>
<td>What is used in the natural recursion on a list?</td>
<td><code>(cdr lat)</code></td>
</tr>
<tr>
<td>What is used in the natural recursion on a vec?</td>
<td><code>(cdr vec)</code></td>
</tr>
<tr>
<td>Why?</td>
<td>Because the rest of a non-empty list is a list and the rest of a non-empty vec is a vec</td>
</tr>
<tr>
<td>How many questions do we need to ask about a list?</td>
<td>Two</td>
</tr>
<tr>
<td>How many questions do we need to ask about a vec?</td>
<td>Two, because it is either empty or it is a number and a rest, which is again a vec</td>
</tr>
</tbody>
</table>
How many questions do we need to ask about a number? Two

The Fourth Commandment

(\textit{preliminary})

When recurring on a list of atoms, \textit{lat}, or a vec, \textit{vec}, ask two questions about them, and use (cdr \textit{lat}) or (cdr \textit{vec}) for the natural recursion.

When recurring on a number, \textit{n}, ask two questions, and use (sub1 \textit{n}) for the natural recursion.

What does cons do? It builds lists

What does addvec do? It builds a number by totalling all the numbers in a vec

What is the terminal condition line of addvec \((\text{(null? vec) 0})\)

What is the natural recursion for addvec \((\text{addvec (cdr vec)})\)

What does addvec use to build a number? It uses \textbf{+} because \textbf{+} \textit{builds numbers}
What will be the last line in the function

```
(define addvec
    (lambda (vec)
        (cond
            ((null? vec) 0)
            (t __________))))
```

(t (+ (car vec) (addvec (cdr vec))))

Notice the similarity between this line, and the last line of the function `rememb`:

(t (cons (car lat) (rememb a (cdr lat))))

What is \(x \times n \times m\), where

\(n\) is 5, and
\(m\) is 3

What is \(x \times n \times m\), where

\(n\) is 1.3, and
\(m\) is 4

What does \(x \times n \times m\) do? It builds up a number by adding \(n\) up \(m\) times

What is the terminal condition line for \(x\)? \((\text{zero? } m) 0\), because \(n \times 0 = 0\)

Since \((\text{zero? } m)\) is the terminal condition, \(m\) must eventually be reduced to zero. What function is used to do this?

Sub1

What is another name for \((x \times n \times \text{sub1 } m)\) in this case? Natural recursion

Try to write the function \(x\)

```
(define x
    (lambda (n m)
        (cond
            ((zero? m) 0)
            (t (+ n (x n (sub1 m)))))))
```
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the meaning of ((+ n \times n (\text{sub1}\ m)))</td>
<td>It adds (n), that is 12, to the natural recursion. If (\times) is correct then ((\times 12 (\text{sub1}\ 3))) should be 24</td>
</tr>
<tr>
<td>What are the new arguments of ((\times n\ m))</td>
<td>(n) is 12, and (m) is 2</td>
</tr>
<tr>
<td>((\text{zero?}\ m))</td>
<td>No</td>
</tr>
<tr>
<td>What is the meaning of ((+ n \times n (\text{sub1}\ m)))</td>
<td>It adds (n), that is 12, to ((\times n (\text{sub1}\ m)))</td>
</tr>
<tr>
<td>What are the new arguments of ((\times n\ m))</td>
<td>(n) is 12, and (m) is 1</td>
</tr>
<tr>
<td>((\text{zero?}\ m))</td>
<td>No</td>
</tr>
<tr>
<td>What is the meaning of ((+ n \times n (\text{sub1}\ m)))</td>
<td>It adds (n), that is 12, to ((\times n (\text{sub1}\ m)))</td>
</tr>
<tr>
<td>What is the value of the line (((\text{zero?}\ m)\ 0))</td>
<td>0, because ((\text{zero?}\ m)) is now true</td>
</tr>
<tr>
<td>Are we finished yet?</td>
<td>No</td>
</tr>
</tbody>
</table>
Argue, using equations, that \( x \) is the conventional multiplication of nonnegative integers, where \( n \) is 12 and \( m \) is 3

\[
egin{align*}
12 \times 3 &= 12 \times 2 + 12 \\
12 \times 2 &= 12 \times 1 + 12 \\
12 \times 1 &= 12 \times 0 + 12 \\
12 \times 0 &= 0 \\
12 \times 3 &= 0 + 12 + 12 + 12 \\
&= 36
\end{align*}
\]

Which is as we expected. This technique works for all recursive functions, not just those that use numbers. You can use this approach to write functions as well as to argue their correctness.

Again, why is 0 the value for the terminal condition line in \( \times \)

Because 0 will not affect \( + \)

That is,

\[
 n + 0 = n
\]

---

**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.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is ((\text{vec}+ \text{vec1} \text{vec2})), where \text{vec1} is ((3 6 9 11 4)), and \text{vec2} is ((8 5 2 0 7))</td>
<td>((11 11 11 11 11))</td>
</tr>
<tr>
<td>What is ((\text{vec}+ \text{vec1} \text{vec2})), where \text{vec1} is ((2 3)), and \text{vec2} is ((4 6))</td>
<td>((6 9))</td>
</tr>
<tr>
<td>What does ((\text{vec}+ \text{vec1} \text{vec2})) do?</td>
<td>It adds the first number of \text{vec1} to the first number of \text{vec2}, then it adds the second number of \text{vec1} to the second number of \text{vec2}, and so on, building a vec of the answers, for vecs of the same length.</td>
</tr>
<tr>
<td>What is unusual about (\text{vec}+)</td>
<td>It looks at each element of two vecs at the same time, or in other words, it recurs on two vecs.</td>
</tr>
<tr>
<td>If you recur on one vec how many questions do you have to ask?</td>
<td>Two, they are (null? vec) and t</td>
</tr>
<tr>
<td>When recurring on two vecs, how many questions need to be asked about the vecs?</td>
<td>Four: if the first vec is empty or non-empty, and if the second vec is empty or non-empty</td>
</tr>
<tr>
<td>Can the first vec be ((\ )) at the same time as the second is other than ((\ ))</td>
<td>No, because the vecs are of the same length</td>
</tr>
<tr>
<td>How many questions do we really need?</td>
<td>Two</td>
</tr>
</tbody>
</table>
Write the function \texttt{vec+}

\begin{verbatim}
(define vec+
  (lambda (vec1 vec2)
    (cond
      ((null? vec1) (quote ( )))
      (t (cons (+ (car vec1) (car vec2))
           (vec+ (cdr vec1) (cdr vec2)))))))
\end{verbatim}

What are the arguments of \texttt{+} in the last line? \texttt{(car vec1) and (car vec2)}

What are the arguments of \texttt{cons} in the last line? \texttt{(+ (car vec1) (car vec2))}, and \texttt{(vec+ (cdr vec1) (cdr vec2))}

What is \texttt{(vec+ vec1 vec2)}, where \texttt{vec1} is \texttt{(3 7)}, and \texttt{vec2} is \texttt{(4 6)}? \texttt{(7 13)}. But let's see how it works.

\texttt{(null? vec1)} No

\texttt{(cons (+ (car vec1) (car vec2))}
\texttt{(vec+ (cdr vec1) (cdr vec2)))}

Cons 7 onto the natural recursion \texttt{(vec+ (cdr vec1) (cdr vec2))}.

Why does the natural recursion include the \texttt{cdr} of both arguments? Because the typical element of the final value uses the car of \texttt{both vecs}, so now we are ready to consider the rest of \texttt{both vecs}.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the value of the line (((\text{null? vec1}) (\text{quote ( ))})))</td>
<td>( )</td>
</tr>
<tr>
<td>What is the value of the application?</td>
<td>(7 13)</td>
</tr>
<tr>
<td>That is, the cons of 7 onto the cons of 13 onto ( )</td>
<td></td>
</tr>
<tr>
<td>What problem arises when we want ((\text{vec+ vec1 vec2})), where (\text{vec1}) is (3 7), and (\text{vec2}) is (4 6 8 1)</td>
<td>When (\text{vec1}) eventually gets to be ( ), we quit, but that means the final value will be (7 13), which is wrong. The final value should be (7 13 8 1)</td>
</tr>
<tr>
<td>Can we still write (\text{vec+}) even if the vecs are not the same length?</td>
<td>Yes!</td>
</tr>
<tr>
<td>What trivial change can we make in the terminal condition line to get the correct final value?</td>
<td>Replace (((\text{null? vec1}) (\text{quote ( ))}))) by (((\text{null? vec1}) \text{vec2}))</td>
</tr>
<tr>
<td>What is ((\text{vec+ vec1 vec2})), where (\text{vec1}) is (3 7 8 1), and (\text{vec2}) is (4 6)</td>
<td>No answer, since (\text{vec2}) will become null before (\text{vec1}). See The Fourth Commandment. We did not ask all the necessary questions!</td>
</tr>
<tr>
<td>What do we need to include in our function?</td>
<td>Another terminal condition</td>
</tr>
</tbody>
</table>
What is the other terminal condition line?

\[(\text{null? vec2})\text{ vec1}\]

So now that we have expanded our function definition so that vec+ works for any two vecs, see if you can rewrite it:

\[
\begin{align*}
\text{(define vec+} \\
\quad (\text{lamb} \text{da} \text{ (vec1 vec2)} \\
\quad \text{(cond} \\
\quad \quad ((\text{null? vec1}) \text{ vec2}) \\
\quad \quad ((\text{null? vec2}) \text{ vec1}) \\
\quad \quad (\text{t} \text{ (cons (+ (car vec1) (car vec2))} \\
\quad \quad \text{(vec+} \\
\quad \quad \text{(cdr vec1) (cdr vec2)})))))
\end{align*}
\]

Does the order of the two terminal conditions matter?

No

Is t really the last question?

Yes, because either \((\text{null? vec1})\) or \((\text{null? vec2})\) is true if either one of them does not contain at least one number.

What is \((> n m)\) where \(n\) is 12, and \(m\) is 13

\(\text{nil, that is, false}\)

What is \((> n m)\) where \(n\) is 12, and \(m\) is 11

\(\text{t, that is, true}\)

On how many numbers do we have to recur?

Two, \(n\) and \(m\)
How many questions do we have to ask about \( n \) and \( m \)?

Three \((\text{zero? } n), (\text{zero? } m), \text{ and } t\)

Can you write the function \( > \) now using \text{zero?}, \text{add1}, \text{and sub1}?

How about

\[
\text{(define >)} \\
\text{(lambda (n m)} \\
\text{(cond)} \\
\text{((zero? m) t)} \\
\text{((zero? n) nil)} \\
\text{(t (> (sub1 n) (sub1 m)))))}}
\]

Is the way we wrote \((> n m)\) correct?

No, try it for the case where \( n \) and \( m \) are the same number. Let \( n \) and \( m \) be 3

\((\text{zero? } m), \text{ where } n \text{ is 3, and } m \text{ is 3}\)

No, so move to the next question

\((\text{zero? } n), \text{ where } n \text{ is 3, and } m \text{ is 3}\)

No, so move to the next question

What is the meaning of \((> (\text{sub1 } n) (\text{sub1 } m))\)

Recur, but with both arguments reduced by one.

\((\text{zero? } m), \text{ where } n \text{ is 2, and } m \text{ is 2}\)

No, so move to the next question
What is the meaning of $(> (sub1 n) (sub1 m))$

Recur, but with both arguments closer to zero by one

$(zero? m)$, where
$n$ is 1, and
$m$ is 1

No, so move to the next question

$(zero? n)$, where
$n$ is 1, and
$m$ is 1

No, so move to the next question

What is the meaning of $(> (sub1 n) (sub1 m))$

Recur, but with both arguments reduced by one

$(zero? m)$, where
$n$ is 0, and
$m$ is 0

Yes, so the value of $(> n m)$ is $t$

Is this correct?

No because 3 is not greater than 3

How can we change the function $>$ to take care of this subtle problem?

Switch the zero? lines, that is

```
(define >
  (lambda (n m)
    (cond
     ((zero? n) nil)
     ((zero? m) t)
     (t (> (sub1 n) (sub1 m)))))))
```
(< n m), where
m is 6

Now try to write <

(define <
(lambda (n m)
  (cond
    ((zero? m) nil)
    ((zero? n) t)
    (t (< (sub1 n) (sub1 m))))))

Here is the definition of =

(define =
  (lambda (n m)
    (cond
     ((zero? m) (zero? n))
     ((zero? n) nil)
     (t (= (sub1 n) (sub1 m)))))

Rewrite = using < and >

Does this mean we have two different functions for testing equality of atoms?

Yes, they are = for atoms which are numbers and eq? for the others.¹

¹ In practice, eq? does work for some numbers

(\n m), where
m is 1

1
(↑ n m), where
n is 2, and
m is 3

(↑ n m), where
n is 5, and
m is 3

Now write the function ↑
Hint: See the Fourth and Fifth Commandments

\[
\text{(define ↑)}
\text{\hspace{1em}}
\text{(lambda (n m)}
\text{(cond)
\text{\hspace{1em}}
\text{((zero? m) 1)}
\text{\hspace{1em}}
\text{(t (× n (↑ n (sub1 m))))})}
\]

What is the value of (length lat), where
lat is (hotdogs with mustard sauerkraut and pickles)

What is (length lat), where
lat is (ham and cheese on rye)

Now try to write the function length

\[
\text{(define length)}
\text{\hspace{1em}}
\text{(lambda (lat)}
\text{(cond)
\text{\hspace{1em}}
\text{((null? lat) 0)}
\text{\hspace{1em}}
\text{(t (add1 (length (cdr lat))))})}
\]

What is (pick n lat), where
n is 4, and
lat is (lasagna spaghetti ravioli macaroni meatball)

Numbers Games
What is \( \text{pick } n \text{ lat} \), where 
\( n \) is 0, and 
\( \text{lat} \) is ( )?

Let's define one \( \text{nil} \)

Try to write the function \text{pick}:

\[
(\text{define } \text{pick} \\
(\lambda (n \text{ lat}) \\
(\text{cond} \\
((\text{null? } \text{lat}) \text{ nil}) \\
((\text{zero? } (\text{sub1 } n)) (\text{car } \text{lat})) \\
(t (\text{pick } (\text{sub1 } n) (\text{cdr } \text{lat}))))))
\]

Does the order of the two terminal conditions matter?

Think about it

Does the order of the two terminal conditions matter?

Try it out!

Does the order of the two previous answers matter?

Yes Think first, then try

What is \( \text{rempick } n \text{ lat} \), where
\( n \) is 3, and
\( \text{lat} \) is \( \text{(hotdogs with mustard)} \)

What is \( \text{rempick } n \text{ lat} \), where
\( n \) is 0, and
\( \text{lat} \) is ( )

Let's define an answer ( )
Now try to write rempick

\[
\text{(define rempick (lambda } (n \ \text{lat})
\text{ (cond}
\text{ ((null? \ \text{lat}) (quote ( ))))
\text{ ((zero? (sub1 \ n)) (cdr \ \text{lat}))
\text{ (t (cons (car \ \text{lat})
\text{ (rempick (sub1 \ n) (cdr \ \text{lat)))))))))}
\]

Is \(\text{(number?}^1 \ a)\) true or false, where \(a\) is \text{tomato} \hspace{1cm} \text{False}

\(1\) \text{ number?}

Is \(\text{(number?} \ a)\) true or false, where \(a\) is \text{76} \hspace{1cm} \text{True}

Can you write \text{number?}, which is true if its argument is a numeric atom and false if its argument is a non-numeric atom? \hspace{1cm} \text{No number?}, like \text{add1 sub1 zero?, car, cdr, cons, null?, eq? and atom?}, is a primitive function.

Now, using \text{number?}, write a function \text{no-nums}, which gives as a final value a \text{lat} obtained by removing all the numbers from the \text{lat}. For example, where \text{lat} is
(\text{5 pears 6 prunes 9 dates}),
(\text{no-nums \text{lat}}) is
(\text{pears prunes dates})

\[
\text{(define no-nums (lambda } (\text{lat})
\text{ (cond}
\text{ ((null? \ \text{lat}) (quote ( ))))
\text{ (t (cond}
\text{ ((number? (car \ \text{lat}))
\text{ (no-nums (cdr \ \text{lat})))
\text{ (t (cons (car \ \text{lat})
\text{ (no-nums (cdr \ \text{lat})))))))})}
\]

Now write all nums which builds a vec as a final value given a lat

```
(define all-nums
  (lambda (lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
          ((number? (car lat))
           (cons (car lat)
                (all-nums (cdr lat))))
          (t (all-nums (cdr lat))))))))
```

Write the function eqan? which is true if its two arguments, a1 and a2, are the same atom. Remember to use = for numbers and eq? for all others

```
(define eqan?
  (lambda (a1 a2)
    (cond
      ((number? a1)
       (cond
        ((number? a2) (= a1 a2))
        (t nil))
      ((number? a2) nil)
      (t (eq? a1 a2))))))
```

Can we assume that all functions written using eq? can be generalized by replacing eq? by eqan? Yes, except of course, for eqan? itself
Wouldn't a (ham and cheese on rye) be goo

Don't forget the mustard!
Exercises

For these exercises,

\[ vec1 \text{ is } (1 \ 2) \]
\[ vec2 \text{ is } (3 \ 2 \ 4) \]
\[ vec3 \text{ is } (2 \ 1 \ 3) \]
\[ vec4 \text{ is } (6 \ 2 \ 1) \]
\[ l \text{ is } () \]
\[ zero \text{ is } 0 \]
\[ one \text{ is } 1 \]
\[ three \text{ is } 3 \]
\[ obj \text{ is } (x \ y) \]

4.1 Write the function duplicate of \( n \) and \( obj \) which builds a list containing \( n \) objects \( obj \).

Example: (duplicate \( three \) \( obj \)) is \((x \ y) \ (x \ y) \ (x \ y)\),
(duplicate \( zero \) \( obj \)) is \( () \),
(duplicate \( one \) \( vec1 \)) is \((1 \ 2)\)

4.2 Write the function \( \text{multvec} \) that builds a number by multiplying all the numbers in a.

Example: (multvec \( vec2 \)) is 24,
(multvec \( vec3 \)) is 6,
(multvec \( l \)) is 1

4.3 When building a value with \( \top \), which value should you use for the terminal line?

4.4 Argue the correctness for the function \( \top \) as we did for \((\times \ m \ n)\). Use 3 and 4 as data.
4.5 Write the function index of a and lat that returns the place of the atom a in lat. You may assume that a is a member of lat. Hint: Can lat be empty?

Example: When a is car,

lat1 is (cons cdr car null? eq?),

b is motor, and

lat2 is (car engine auto motor),

we have (index a lat1) is 3,

(index a lat2) is 1,

(index b lat2) is 4

4.6 Write the function product of vec1 and vec2 that multiplies corresponding numbers in vec1 and vec2 and builds a new vec from the results. The vecs, vec1 and vec2, may differ in length.

Example: (product vec1 vec2) is (3 4 4),

(product vec2 vec3) is (6 2 12),

(product vec3 vec4) is (12 2 3)

4.7 Write the function dot product of vec1 and vec2 that multiplies corresponding numbers in vec1 and vec2 and builds a new number by summing the results. The vecs, vec1 and vec2, are the same length.

Example: (dot-product vec2 vec2) is 29,

(dot-product vec2 vec4) is 26,

(dot-product vec3 vec4) is 17

4.8 Write the function / that divides nonnegative integers.

Example: (/ n m) is 1, when n is 7 and m is 5.

(/ n m) is 4, when n is 8 and m is 2

(/ n m) is 0, when n is 2 and m is 3

Hint: A number is now defined as a rest (between 0 and m - 1) and a multiple addition of m. The number of additions is the result.

4.9 Here is the function remainder

```
(define remainder
  (lambda (n m)
    (cond
      (t (– n (× m (/ n m)))))))
```

Make up examples for the application (remainder n m) and work through them.

4.10 Write the function ≤ which tests if two numbers are equal or if the first is less than the second.

Example: (≤ zero one) is true,

(≤ one one) is true,

(≤ three one) is false
The Multichapter Chapter
Write the function member?

(defun member?
  (lambda (a lat)
    (cond
      ((null? lat) nil)
      (t (or
        (eq? (car lat) a)
        (member? a (cdr lat))))))

Do you recall, or can you see now what member? does?

It checks each atom of the lat to see if it is the same as the atom a. When it finds the first occurrence of a, it stops and returns t

Write the function rember

(defun rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ( )))
      (t (cond
        ((eq? (car lat) a) (cdr lat))
        (t (cons (car lat)
          (rember
            a (cdr lat)))))))))

Do you recall or can you see now, what rember does?

Rember looks at each atom of the lat to see if it is the same as the atom a. If it is not rember saves the atom and proceeds. When it finds the first occurrence of a, it stops and gives the value (cdr lat), or the rest of the lat, so that the value returned is the original lat, with only the first occurrence of a removed
Write the function multirember which gives as its final value the lat with all occurrences of a removed

\[
\text{(define multirember} \\
\text{ (lambda (a lat))} \\
\text{ (cond} \\
\text{ (________ _______ }) \\
\text{ (t (cond} \\
\text{ (________ _______ })} \\
\text{ (________ _______ })))))
\]

Hint: What do we want as the value when (eq? (car lat) a) is true?
Consider the example where a is cup, and lat is (coffee cup tea cup and hick cup)

After the first occurrence of a, we now recur with (multirember a (cdr lat)), in order to remove the other occurrences.

The value of the application is (coffee tea and hick)

Can you see how multirember works?
Possibly not, so we will go through the steps necessary to arrive at the value (coffee tea and hick)

\[(\text{null? lat})\]

No, so move to the next line

\[t\]

\[(\text{eq? (car lat) a})\]

No, so move to the next line

What is the meaning of (cons (car lat) (multirember a (cdr lat)))

Save (car lat), namely coffee, to be consed onto the value of (multirember a (cdr lat)) later. Now determine (multirember a (cdr lat))
<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(eq? (car lat) a)</td>
<td>Yes, so forget (car lat), and determine (multirember a (cdr lat)).</td>
</tr>
<tr>
<td>(null? lat)</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>t</td>
<td>Yes¹</td>
</tr>
<tr>
<td>(eq? (car lat) a)</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>(cons</td>
<td>Save (car lat), namely tea, to be consed onto the value of (multirember a (cdr lat)) later</td>
</tr>
<tr>
<td>(car lat)</td>
<td>Now determine (multirember a (cdr lat)).</td>
</tr>
<tr>
<td>(multirember a (cdr lat))</td>
<td></td>
</tr>
<tr>
<td>(null? lat)</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>(eq? (car lat) a)</td>
<td>Yes, so forget (car lat), and determine (multirember a (cdr lat)).</td>
</tr>
<tr>
<td>(null? lat)</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>(eq? (car lat) a)</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>Question</td>
<td>Answer</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>What is the meaning of ((\text{cons} (\text{car } \text{lat}) (\text{multirember } a (\text{cdr } \text{lat})))))</td>
<td>Save ((\text{car } \text{lat})), namely (\text{hick}), to be consed onto the value of ((\text{multirember } a (\text{cdr } \text{lat})))) later. Now determine ((\text{multirember } a (\text{cdr } \text{lat}))))</td>
</tr>
<tr>
<td>((\text{null? } \text{lat}))</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>((\text{eq? } (\text{car } \text{lat}) \ a))</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>What is the meaning of ((\text{cons} (\text{car } \text{lat}) (\text{multirember } a (\text{cdr } \text{lat})))))</td>
<td>Save ((\text{car } \text{lat})), namely (\text{hick}), to be consed onto the value of ((\text{multirember } a (\text{cdr } \text{lat})))) later. Now determine ((\text{multirember } a (\text{cdr } \text{lat}))))</td>
</tr>
<tr>
<td>((\text{null? } \text{lat}))</td>
<td>No, so move to the next line</td>
</tr>
<tr>
<td>((\text{eq? } (\text{car } \text{lat}) \ a))</td>
<td>Yes, so forget ((\text{car } \text{lat})), and determine ((\text{multirember } a (\text{cdr } \text{lat})))).</td>
</tr>
<tr>
<td>((\text{null? } \text{lat}))</td>
<td>Yes, so we have a value of ((\text{ )}))</td>
</tr>
<tr>
<td>Are we finished?</td>
<td>No, we still have several conses to pick up</td>
</tr>
<tr>
<td>What do we do next?</td>
<td>We cons the most recent ((\text{car } \text{lat})) we have, namely (\text{hick}), onto ((\text{ )}))</td>
</tr>
</tbody>
</table>
What do we do next? We cons coffee onto (tea and hick)

Are we finished now? Yes

Now write the function multiinsertr

\[
\text{(define multiinsertr)} \\
\text{(lambda (new old lat))} \\
\text{(cond}} \\
\text{((null? lat) (quote ()))} \\
\text{(t (cond}} \\
\text{((eq? (car lat) old) (cons (car lat) (cons new (multiinsertr new old (cdr lat))))))} \\
\text{(t (cons (car lat) (multiinsertr new old (cdr lat)))))))}
\]

It would also be correct to use old in place of (car lat) because we know that (eq? (car lat) old)
Is this function defined correctly?

\[\text{(define multiinsertL} \\text{(lambda (new old lat)}} \\text{(cond} \]
\[\text{((null? lat) (quote ( )))} \]
\[\text{(t (cond} \]
\[\text{((eq? (car lat) old)} \]
\[\text{(cons new} \]
\[\text{(cons old} \]
\[\text{(multiinsertL} \]
\[\text{new old lat)))})\]
\[\text{(t (cons} \]
\[\text{(car lat)} \]
\[\text{(multiinsertL} \]
\[\text{new old (cdr lat))))))))\]
\[\text{)}\text{)}\]

Not quite. To find out why, go through the function, where
\text{new} is fried,
\text{old} is fish, and
\text{lat} is (chips and fish or fish and fried)

Was the terminal condition ever reached?

No, because we never get past the first occurrence of \text{old}.

Now try to write the function \text{multiinsertL}

\[\text{(define multiinsertL} \\text{(lambda (new old lat)}} \\text{(cond} \]
\[\text{((null? lat) (quote ( )))} \]
\[\text{(t (cond} \]
\[\text{((eq? (car lat) old)} \]
\[\text{(cons new} \]
\[\text{(cons old} \]
\[\text{(multiinsertL} \]
\[\text{new old lat)))})\]
\[\text{(t (cons (car lat)} \]
\[\text{)}\text{)}\]
\[\text{(multiinsertL} \]
\[\text{new old (cdr lat))))))))\]
\[\text{)}\text{)}\]
The Sixth Commandment

Always change at least one argument while recurring. The changing argument(s) should be tested in the termination condition(s) and it should be changed to be closer to termination. For example:

When using cdr, test termination with null?
When using subl, test termination with zero?

Now write the function multisubst

```
(define multisubst
 (lambda (new old lat)
   (cond
    (_____ _____)
    (t (cond
        (_____ _____)
        (_____ _____))))))
```

Now write the function occur which counts the number of times an atom \( a \) appears in a \( lat \).

```
(define occur
 (lambda (a lat)
   (cond
    ((null? lat) 0)
    (t (cond
        ((eq? (car lat) a)
         (addl (occur a (cdr lat))))
        (t (occur a (cdr lat))))))))
```
Write the function `one?` where `(one? n)` is true if `n` is 1, and false (i.e., `nil`) otherwise.

```
(define one?
  (lambda (n)
    (cond
      ((zero? n) nil)
      (t (zero? (sub1 n))))))
```

or

```
(define one?
  (lambda (n)
    (cond
      (t (= n 1))))
```

---

Guess how we can further simplify this function, making it a one liner.

By removing the `(cond` clause, we get

```
(define one?
  (lambda (n)
    (= n 1))
```

---

Now rewrite the function `rempick` that removes the `n`th atom from the `lat`. For example, where

- `n` is 3, and
- `lat` is `lemon meringue salty pie`.

the application `(rempick n lat)` has the value `lemon-meringue pie`.

Use the function `one?` in your answer.

```
(define rempick
  (lambda (n lat)
    (cond
      ((null? lat) (quote ( )))
      ((one? n) (cdr lat))
      (t (cons (car lat)
                (rempick
                 (sub1 n) (cdr lat)))))))
```

---

Is `rempick` a "multi" function? No
Exercises

For these exercises,

\[ \begin{align*}
\text{x is} & \quad \text{comma} \\
\text{y is} & \quad \text{dot} \\
\text{a is} & \quad \text{kiwis} \\
\text{b is} & \quad \text{plums} \\
\text{lat1 is} & \quad \text{bananas kiwis} \\
\text{lat2 is} & \quad \text{peaches apples bananas} \\
\text{lat3 is} & \quad \text{kiwis pears plums bananas cherries} \\
\text{lat4 is} & \quad \text{kiwis mangoes kiwis guavas kiwis} \\
\text{l1 is} & \quad \text{((curry) ( ) (chicken) ( ) )} \\
\text{l2 is} & \quad \text{((peaches) (and cream))} \\
\text{l3 is} & \quad \text{((plums) and (ice) and cream)} \\
\text{l4 is} & \quad \text{()} \\
\end{align*} \]

5 1. For Exercise 3.4 you wrote the function subst cake. Write the function multisubst.

Example: 

\[
\begin{align*}
\text{multisubst-kiwis b lat1} & \quad \text{is} \text{ (bananas plums),} \\
\text{multisubst-kiwis y lat2} & \quad \text{is} \text{ (peaches apples bananas),} \\
\text{multisubst-kiwis y lat4} & \quad \text{is} \text{ (dot mangoes dot guavas dot),} \\
\text{multisubst kiwis y l4} & \quad \text{is} \text{ ( )} \\
\end{align*}
\]

5 2. Write the function multisubst2. You can find subst2 at the end of Chapter 3.

Example: 

\[
\begin{align*}
\text{multisubst2 x a b lat1} & \quad \text{is} \text{ (bananas comma),} \\
\text{multisubst2 y a b lat2} & \quad \text{is} \text{ (dot pears dot bananas cherries),} \\
\text{multisubst2 a x y lat1} & \quad \text{is} \text{ (bananas kiwis).} \\
\end{align*}
\]

5 3. Write the function multidown of lat which replaces every atom in lat by a is the atom.

Example: 

\[
\begin{align*}
\text{multidown lat1} & \quad \text{is} \text{ ((bananas) (kiwis)),} \\
\text{multidown lat2} & \quad \text{is} \text{ ((peaches) (apples) (bananas)),} \\
\text{multidown l4} & \quad \text{is} \text{ ( )} \\
\end{align*}
\]

The Multichapter Chapter
5.4 Write the function occurN of alat and lat which counts how many times an atom also occurs in lat

Example: (occurN lat1 lat) is 0,
(occurN lat1 lat2) is 1,
(occurN lat1 lat3) is 2

5.5 The function I of lat1 and lat2 returns the first atom in lat2 that is in both lat1 and lat2.

Write the functions I and multI. multI returns a list of atoms common to lat1 and lat2.

Example: (I lat1 lat) is ( ),
(I lat1 lat2) is bananas,
(I lat1 lat3) is kiwis;
(multiI lat1 lat) is ( ),
(multiI lat1 lat2) is (bananas),
(multiI lat1 lat3) is (kiwis bananas)

5.6 Consider the following alternative definition of one?

(define one?
  (lambda (n)
    (cond
      ((zero? (sub1 n)) t)
      (t nil)))))

Which Laws and/or Commandments does it violate?

5.7 Consider the following definition of =

(define =
  (lambda (n m)
    (cond
      ((zero? n) (zero? m))
      (t (= n (sub1 m))))))

This definition violates The Sixth Commandment. Why?
5.8 The function count0 of vec counts the number of zero elements in vec. What is wrong with the following definition? Can you fix it?

```
(define count0
  (lambda (vec)
    (cond
      ((null? vec) 1)
      (t (cond
          ((zero? (car vec))
            (cons 0 (count0 (cdr vec))))
          (t (count0 (cdr vec)))))))
```

5.9 Write the function multiup of l which replaces every list of length one in l by the atom in that list, and which also removes every empty list.

Example: (multiup 4) is ( ),
           (multiup (11) is (curry chicken),
           (multiup (12) is (peaches (and cream)))

5.10 Review all the Laws and Commandments. Check the functions in Chapters 4 and 5 to see if they obey the Commandments. When did we not obey them literally? Did we act according to their spirit?

---

But answer came there none—
And this was scarcely odd, because
They'd eaten every one

The Walrus and The Carpenter

—Lewis Carroll
*Oh My Gawd*:
It’s Full of Stars
True or false, \((\text{not (atom? s)})\), where 
\(s\) is (hungarian goulash) 

\((\text{not (atom? s)})\) where 
\(s\) is atom

\((\text{not (atom? s)})\), where 
\(s\) is (turkish ((coffee) and) baklava) 

What is (leftmost \(l\)), where 
\(l\) is ((hot) (tuna (and)) cheese) 

(lat? \(l\)), where 
\(l\) is ((hot) (tuna (and)) cheese) 

Is (car \(l\)) an atom, where 
\(l\) is ((hot) (tuna (and)) cheese) 

What is (leftmost \(l\)), where 
\(l\) is (((hamburger) french) 
(fries (and a) coke)) 

What is (leftmost \(l\)), where 
\(l\) is (((((4 four)) 17 (seventeen)))
Write \textbf{occur*}:

\begin{verbatim}
(define occur*
  (lambda (a l)
    (cond
      ((null? l) 0)
      ((non-atom? (car l))
       (+ (occur* a (car l))
          (occur* a (cdr l))))
      (t (cond
          ((eq? (car l) a)
           (add1 (occur* a (cdr l))))
          (t (occur* a (cdr l)))))))
\end{verbatim}

(subst* new old l) where
\textbf{new} is orange,
\textbf{old} is banana, and
\textbf{l} is ((banana)
  (split (((banana ice)))
    (cream (banana))
    sherbet))
  (banana)
  (bread)
  (banana brandy))
Write subst*:

\[
\text{(define subst* (lambda (new old l)}
\begin{align*}
&\text{(cond} \\
&\quad (\text{______ ________)}) \\
&\quad (\text{______ ________)}) \\
&\quad (\text{______ ________)}) \\
\end{align*}
\]

\[
\text{(define subst* (lambda (new old l)}
\begin{align*}
&\text{(cond} \\
&\quad ((\text{null? l}) (\text{quote ())))} \\
&\quad ((\text{non atom? (car l)}) \\
&\qquad (\text{cons} \\
&\qquad \quad (\text{subst* new old (car l)})} \\
&\qquad \quad (\text{subst* new old (cdr l)})))} \\
&\quad (t (\text{cond} \\
&\qquad ((\text{eq? (car l) old)} \\
&\qquad \quad (\text{cons new} \\
&\qquad \quad \quad (\text{subst* new old (cdr l)})))} \\
&\qquad (t (\text{cons (car l)} \\
&\qquad \quad (\text{subst* new old (cdr l)})))))) \\
\end{align*}
\]

What is (insertL* new old l), where
new is pecker,
old is chuck, and
l is ((how much (wood))
could
((a (wood) chuck))
(((chuck)))
(if (a) ((wood chuck)))
could chuck wood)

\[
((\text{how much (wood)})) \\
\text{could} \\
((\text{a (wood) pecker chuck})) \\
(((\text{pecker chuck}))) \\
((\text{if (a) ((wood pecker chuck))}) \\
\text{could pecker chuck wood).}
\]
Write insertL*:

\[
\begin{align*}
&\text{(define insertL*} \\
&\quad \text{(lambda} \ (\text{new} \ \text{old} \ \text{l}) \\
&\quad \quad \text{(cond}} \\
&\quad \quad \quad \text{)} \\
&\quad \quad \quad \text{)} \\
&\quad \quad \text{)} \\
&\quad \text{)))}) \\
&\end{align*}
\]

\[
\begin{align*}
&(\text{define insertL*} \\
&\quad \text{(lambda} \ (\text{new} \ \text{old} \ \text{l}) \\
&\quad \quad \text{(cond}} \\
&\quad \quad \quad \text{))} \\
&\quad \quad \text{)} \\
&\quad \text{)} \\
&\quad \text{)))}) \\
&\end{align*}
\]

(member* a l), where

\begin{align*}
&\text{a is chips, and} \\
&\text{l is ((potato) (chips ((with) fish) (chips)))} \\
&\end{align*}

t because the atom chips appears in the list l

Write member*:

\[
\begin{align*}
&\text{(define member*} \\
&\quad \text{(lambda} \ (\text{a} \ \text{l}) \\
&\quad \quad \text{(cond}} \\
&\quad \quad \quad \text{)} \\
&\quad \quad \text{)} \\
&\quad \quad \text{)} \\
&\quad \text{))))}) \\
&\end{align*}
\]

\[
\begin{align*}
&(\text{define member*} \\
&\quad \text{(lambda} \ (\text{a} \ \text{l}) \\
&\quad \quad \text{(cond}} \\
&\quad \quad \quad \text{))} \\
&\quad \quad \text{)} \\
&\quad \text{)} \\
&\quad \text{))))}) \\
&\end{align*}
\]
What is \((\text{member}^* \ a \ l)\), where
\(a\) is chips, and
\(l\) is \(\text{(potato) (chips ((with) fish) (chips))}\)

Which chips did it find?
\(\text{(potato) (chips ((with) fish) (chips))}\)

Try to write \(\text{member}^*\) without using non-atom?

\[
\begin{aligned}
\text{(define member*} \\
&\quad \text{(lambda} \ (a \ l) \\
&\quad \quad \text{(cond} \\
&\quad \quad \quad ((\text{null?} \ l) \ \text{nil}) \\
&\quad \quad \quad ((\text{atom?} \ (\text{car} \ l)) \\
&\quad \quad \quad \quad \text{(or} \\
&\quad \quad \quad \quad \quad \text{(eq?} \ (\text{car} \ l) \ a) \\
&\quad \quad \quad \quad \quad \quad \text{(member*} \ a \ (\text{cdr} \ l)))))) \\
&\quad \quad \quad \quad \text{(t} \ (\text{or} \\
&\quad \quad \quad \quad \quad \quad \text{(member*} \ a \ (\text{car} \ l) \\
&\quad \quad \quad \quad \quad \quad \text{(member*} \ a \ (\text{cdr} \ l))))))) \\
\end{aligned}
\]

Do you remember what \((\text{or} \ ...\ )\) does?
\((\text{or} \ ...\ )\) asks questions one at a time until it finds one that is true. Then \((\text{or} \ ...\ )\) stops, making its value true. If it cannot find a true argument, then the value of \((\text{or} \ ...\ )\) is false.

What is
\((\text{and} \ (\text{atom?} \ (\text{car} \ l)) \\
\quad \text{(eq?} \ (\text{car} \ l) \ x)))\),
where
\(x\) is pizza, and
\(l\) is \((\text{mozzarella pizza})\)
Why is it false? Since (and ... ) asks (atom? (car l)), and it is not; so it is nil.

Give an example for \( x \) and \( l \) where the expression is true Here's one
\[
x \text{ is pizza, and} \\
l \text{ is } (\text{pizza (tastes good)})
\]

Put in your own words what (and ...) does (and ...) asks questions one at a time until it finds an argument which is false. Then (and ...) stops with false. If it cannot find a false argument, then it is true.

True or false, it is possible that one of the arguments of (and ...) and (or ...) is not considered?\(^1\) True, because (and ...) stops if the first argument has the value nil, and (or ...) stops if the first argument has the value t.

\(^1\) (cond ...) also has the property of not considering all of its arguments.

(eqlist? \( l_1 \ l_2 \)), where \( l_1 \) is (strawberry ice cream) and \( l_2 \) is (strawberry ice cream) \( t \)

(eqlist? \( l_1 \ l_2 \)), where \( l_1 \) is (strawberry ice cream), and \( l_2 \) is (strawberry cream ice) \( \text{nil} \)
(eqlist? l1 l2), where
  l1 is (beef ((sausage)) (and (soda))), and
  l2 is (beef ((salami)) (and (soda)))
  nil, but almost t

(eqlist? l1 l2), where
  l1 is (beef ((sausage)) (and (soda))), and
  l2 is (beef ((sausage)) (and (soda)))
  t  That's better

What is eqlist?

It is a function that determines if the two lists are structurally the same.

Write eqlist? using eqan?

```
(define eqlist?
  (lambda (l1 l2)
    (cond
      ((and (null? l1) (null? l2)) t)
      ((or (null? l1) (null? l2)) nil)
      ((and (non-atom? (car l1))
               (non-atom? (car l2))
             (and (eqlist? (car l1) (car l2))
                  (eqlist? (cdr l1) (cdr l2))))
       (eqlist? (cdr l1) (cdr l2)))
      ((or (non-atom? (car l1))
           (non-atom? (car l2))) nil)
      (t (and
          (eqan? (car l1) (car l2))
          (eqlist? (cdr l1) (cdr l2))))))
```

Why is there no explicit test for atoms?

If we know that the car of each list is not a list, then the car of each list must be an atom.
Write the function equal? which determines if two S expressions are structurally the same.

\[
\text{(define equal?)}
\]  
\[
\text{(lambda (s1 s2))}
\]  
\[
\text{(cond}
\]  
\[
\text{((and (atom? s1) (atom? s2))}
\]  
\[
\text{(eqan? s1 s2))}
\]  
\[
\text{(and}
\]  
\[
\text{(non-atom? s1)}
\]  
\[
\text{(non-atom? s2))}
\]  
\[
\text{(eqlist? s1 s2))}
\]  
\[
\text{(t nil)))}}
\]

Now, rewrite eqlist? using equal?

\[
\text{(define eqlist?)}
\]  
\[
\text{(lambda (ll ll2))}
\]  
\[
\text{(cond}
\]  
\[
\text{((and (null? ll) (null? ll2)) t)}
\]  
\[
\text{((or (null? ll) (null? ll2)) nil)}
\]  
\[
\text{(t (and}
\]  
\[
\text{(equal? (car ll) (car ll2))}
\]  
\[
\text{(eqlist? (cdr ll) (cdr ll2))))))}
\]

Is equal? a “star” function? Yes

How would rember change if we replaced lat by a general list l and if we replaced a by an arbitrary S-expression s?

\[
\text{(define rember)}
\]  
\[
\text{(lambda (s l))}
\]  
\[
\text{(cond}
\]  
\[
\text{((null? l) (quote ()))}
\]  
\[
\text{((non-atom? (car l))}
\]  
\[
\text{(cond}
\]  
\[
\text{((equal? (car l) s) (cdr l))}
\]  
\[
\text{(t (cons (car l)}
\]  
\[
\text{(rember s (cdr l))))))}
\]  
\[
\text{(t (cond}
\]  
\[
\text{((equal? (car l) s) (cdr l))}
\]  
\[
\text{(t (cons (car l)}
\]  
\[
\text{(rember s (cdr l))))))})
\]
And how does that differ? Remember now removes the first matching S expression s in the list l, instead of the first matching atom a in the list l.

Is rember a “star” function now? No

Why not? Because rember only recurs with the (cdr l)

Can you simplify rember? Obviously\(^1\)

\[
\text{(define rember}
\begin{align*}
\text{(lambda (s l)} & \text{ cond} \\
\text{((null? l) (quote ( )))} & \text{ t (cond} \\
& \text{((equal? (car l) s) (cdr l))} \\
& \text{ t (cons (car l)} \\
& \text{ (rember s (cdr l)))))))
\end{align*}
\]

Can you simplify rember even more? Yes, the inner (cond \(\)) is asking questions that the outer (cond \(\)) could ask\(^1\)

Do it\(^1\)

\[
\text{(define rember}
\begin{align*}
\text{(lambda (s l)} & \text{ cond} \\
\text{((null? l) (quote ( )))} & \text{ ((equal? (car l) s) (cdr l))} \\
& \text{ t (cons (car l)} \\
& \text{ (rember s (cdr l)))))))
\end{align*}
\]
Simplify insertL*

```
(define insertL*
  (lambda (new old l)
    (cond
      ((null? l) (quote ( )))
      ((non-atom? (car l))
       (cons
        (insertL* new old (car l))
        (insertL* new old (cdr l)))))
      ((eq? (car l) old)
       (cons new
         (cons old
           (insertL* new old (cdr l)))))
      (t (cons (car l)
               (insertL*
                new old (cdr l))))))
```

Do these new definitions look simpler?
Yes, they do

And they work just as well
Yes, because we know that all the cases and recursions are right before we simplify

The Seventh Commandment
Simplify only after the function is correct.

Can all functions that were written using eq? and = be generalized by replacing eq? and = by the function equal?

Not quite, this won't work for eqan?, but will work for all others. In fact, disregarding the trivial example of eqan?, that is exactly what we shall assume.
For these exercises,

\[ l1 \text{ is } ((\text{fried potatoes}) \ (\text{baked (fried)}) \ \text{tomatoes}) \]
\[ l2 \text{ is } (((\text{chili}) \ \text{chili} \ (\text{chili}))) \]
\[ l3 \text{ is } ( ) \]
\[ \text{lat1 is } (\text{chili and hot}) \]
\[ \text{lat2 is } (\text{baked fried}) \]
\[ a \text{ is fried} \]

\[ l \]

6.1 Write the function down* of \( l \) which puts every atom in \( l \) in a list by itself

Example \((\text{down* } l2) \text{ is } (((((\text{chili}) \ (\text{chili}) \ ((\text{chili}))))), \)
\((\text{down* } l3) \text{ is } ( ) ,\)
\((\text{down* } \text{lat1}) \text{ is } ((\text{chili}) \ (\text{and}) \ (\text{hot})) \)

6.2 Write the function occurN* of \( \text{lat} \) and \( l \) which counts all the atoms that are common to \( \text{lat} \) and \( l \)

Example \((\text{occurN* } \text{lat1 } l2) \text{ is } 3, \)
\((\text{occurN* } \text{lat2 } l1) \text{ is } 3, \)
\((\text{occurN* } \text{lat1 } l3) \text{ is } 0 \)

6.3 Write the function double* of \( a \) and \( l \) which doubles each occurrence of \( a \) in \( l \)

Example \((\text{double* } a \ l1) \text{ is } ((\text{fried fried potatoes}) \ (\text{baked (fried fried)}) \ \text{tomatoes}), \)
\((\text{double* } a \ l2) \text{ is } (((\text{chili}) \ \text{chili} \ (\text{chili}))), \)
\((\text{double* } a \ \text{lat2}) \text{ is } (\text{baked fried fried}) \)

6.4 Consider the function lat? from Chapter 2. Argue why lat? has to ask three questions (and not two like the other functions in Chapter 2). Why does lat? not have to recur on the car?

6.5 Make sure that \((\text{member* } a \ l), \) where

\( a \) is chips and
\( l \) is \(((\text{potato}) \ (\text{chips ((with) fish) (chips)))\),
really discovers the first chips. Can you change member* so that it finds the last chips first?

*Oh My Gawd* It's Full of Stars

111
6.6 Write the function \texttt{hst+} which adds up all the numbers in a general list of numbers

Example: When \texttt{ll} is \(((1 \ (6 \ 6 \ ( ))))\),
and \texttt{ll} is \(((1 \ 2 \ (3 \ 6)) \ 1)\), then
\[
\texttt{(hst+ ll)} \text{ is } 13,
\texttt{(hst+ ll)} \text{ is } 13,
\texttt{(hst+ ll)} \text{ is } 0
\]

6.7 Consider the following function \texttt{g*} of \texttt{hvec} and \texttt{acc}

\begin{verbatim}
(define g*
 (lambda (hvec acc)
   (cond
    ((null? hvec) acc)
    ((atom? (car hvec))
      (g* (cdr hvec) (+ (car hvec) acc)))
    (t (g* (car hvec) (g* (cdr hvec) acc)))))))
\end{verbatim}

The function is always applied to a (general) list of numbers and 0. Make up examples and find out what the function does.

6.8 Consider the following function \texttt{f*} of \texttt{l} and \texttt{acc}

\begin{verbatim}
(define f*
 (lambda (l acc)
   (cond
    ((null? l) acc)
    ((atom? (car l))
      (cond
       ((member? (car l) acc) (f* (cdr l) acc))
       (t (f* (cdr l) (cons (car l) acc)))))
    (t (f* (car l) (f* (cdr l) acc)))))))
\end{verbatim}

The function is always applied to a list and the empty hst. Make up examples for \texttt{l} and step through the applications. Generalize in one sentence what \texttt{f*} does.

6.9 The functions in Exercises 6.7 and 6.8 employ the \textit{accumulator technique}. This means that they pass along an argument that represents the result that has been computed so far. When these functions reach the bottom (null?, zero?), they just return the result contained in the accumulator. The original argument for the accumulator is the element that used to be the answer for the null?-case. Write the function \texttt{occur} (see Chapter 5) of \texttt{a} and \texttt{lat} using the accumulator technique. What is the original value for \texttt{acc}?
6.10 Step through an application of the original occur and the o and compare the arguments in the recursive applications. Can you accumulator technique?

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

*Oh My Gawd* It's Full of Stars
Shadows
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is 1 an arithmetic expression?</td>
<td>Yes</td>
</tr>
<tr>
<td>Is 3 an arithmetic expression?</td>
<td>Yes, of course</td>
</tr>
<tr>
<td>Is 1 + 3 an arithmetic expression?</td>
<td>Yes</td>
</tr>
<tr>
<td>Is 1 + 3 × 4 an arithmetic expression?</td>
<td>Definitely</td>
</tr>
<tr>
<td>Is cookie an arithmetic expression?</td>
<td>Yes Are you almost ready for one?</td>
</tr>
<tr>
<td>And what about 3 ⊕ y ⊕ 5</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| What is an arithmetic expression in your words? | In ours.  
“For the purpose of this chapter, an arithmetic expression is either an atom (including numbers), or two arithmetic expressions combined by ⊕, ×, or ⊕” |
| What is (quote a)                             | a      |
| What is (quote ⊕)                            | ⊕      |
| What does (quote ×) stand for?                | ×      |
Is \((x + 3)\) an arithmetic expression?  

| Not really, since there are parentheses around \(x + 3\). Our definition of arithmetic expression did not mention parentheses. |

Could we think of \((x + 3)\) as an arithmetic expression?  

| Yes, if we keep in mind that the parentheses are not really there. |

What would you call \((x + 3)\)  

| We call it a representation for \(x + 3\). |

Why is \((x + 3)\) a good representation?  

| Because |
| 1. \((x + 3)\) is an S expression. It can therefore serve as an argument for a function, and |
| 2. It structurally resembles the expression we want to represent. |

True or false, \((\text{numbered? } x)\) where \(x\) is 1  

| True |

How do you represent \(3 + 4 \times 5\)?  

| \((3 + (4 \times 5))\) |

True or false, \((\text{numbered? } y)\) where \(y\) is \((3 + (4 \times 5))\)  

| True |

True or false, \((\text{numbered? } z)\) where \(z\) is \((2 \times \text{sausage})\)  

| False, because sausage is not a number. It is a symbol. |
Now you can write a skeleton for numbered?

```
(define numbered?
  (lambda (aexp)
    (cond
      (__________   ________)
      (__________   ________)
      (__________   ________)
      (__________   ________))))
```

is a good guess

What is the first question?

```
(atom? aexp)
```

What is (eq? (car (cdr aexp)) (quote +))

It is the second question

Can you guess the third one?

```
(eq? (car (cdr aexp)) (quote ×)) is perfect
```

And you must know the fourth one

```
(eq? (car (cdr aexp)) (quote ⨿)), of course
```

Should we ask another question about aexp?

No! So we could replace the previous question by ⨿

Why do we ask not two but four questions about arithmetic expressions? After all, arithmetic expressions like (1 + 3) are lats

Because we consider (1 + 3) as a representation of an arithmetic expression in list form, not as a list itself. And, an arithmetic expression is either a number, or two arithmetic expressions combined by +, ×, or ⨿
Now you can almost write numbered?  Here is our proposal

```
(define numbered?
  (lambda (aexp)
    (cond
      ((atom? aexp) (number? aexp))
      ((eq? (car (cdr aexp)) (quote +))
        (eq? (car (cdr aexp)) (quote x))
        (eq? (car (cdr aexp)) (quote t)))
    ))
```

Why do we ask (number? aexp) when we know that aexp is an atom?  Because we want to know if all arithmetic expressions that are atoms are numbers.

What do we need to know if the aexp consists of two arithmetic expressions combined by +?  We need to find out whether the two subexpressions are numbered.

In which position is the first subexpression?  It is the car of aexp.

In which position is the second subexpression?  It is the car of the cdr of the cdr of aexp.

So what do we need to ask?  (numbered? (car aexp)) and (numbered? (car (cdr (cdr aexp))))

Both questions must be true.

What is the second question?  (and (numbered? (car aexp))
  (numbered? (car (cdr (cdr aexp)))))
Try numbered? again

```
(define numbered?
  (lambda (aexp)
    (cond
      ((atom? aexp) (number? aexp))
      ((eq? (car (cdr aexp)) (quote +))
        (and
          (numbered? (car aexp))
          (numbered?
            (car (cdr (cdr aexp)))))))
      ((eq? (car (cdr aexp)) (quote ×))
        (and
          (numbered? (car aexp))
          (numbered?
            (car (cdr (cdr aexp)))))))
      ((eq? (car (cdr aexp)) (quote +))
        (and
          (numbered? (car aexp))
          (numbered?
            (car (cdr (cdr aexp)))))))
  )
```

Since `aexp` is known to be an arithmetic expression, could we have written `numbered?` in a simpler way?

Yes

```
(define numbered?
  (lambda (aexp)
    (cond
      ((atom? aexp) (number? aexp))
      (t (and
          (numbered? (car aexp))
          (numbered?
            (car (cdr (cdr aexp)))))))))
```
(value z) where z is cookie

(value aexp) returns what we think is the natural value of a numbered arithmetic expression

How many questions will value ask about aexp?

Four

Now, let's write a first attempt at value

```
(define value
  (lambda (aexp)
    (cond
      ((number? aexp)
       __________)
      ((eq? (car (cdr aexp)) (quote +))
       __________)
      ((eq? (car (cdr aexp)) (quote *))
       __________)
      (t __________)))))
```

What is the natural value of an arithmetic expression that is a number?

It is just that number

What is the natural value of an arithmetic expression that consists of two arithmetic expressions combined by +

If we had the natural value of the two subexpressions, we could just add up the two values
Can you think of a way to get the value of the two subexpressions in \((1 + (3 \times 4))\) 

Of course, by applying value to 1, and to \((3 \times 4)\)

And in general? 

By recurring with value on the subexpressions.

---

**The Eighth Commandment**

Recur on all the *subparts* that are of the same nature:

— On all the sublists of a list.
— On all the subexpressions of a representation of an arithmetic expression.

---

Give value another try

```
(define value
  (lambda (aexp)
    (cond
      ((number? aexp) aexp)
      ((eq? (car (cdr aexp)) (quote +))
        (+ (value (car aexp))
           (value (car (cdr (cdr aexp))))))
      ((eq? (car (cdr aexp)) (quote \times))
        (\times (value (car aexp))
                 (value (car (cdr (cdr aexp))))))
      (t (value (car aexp))
         (value
          (car (cdr (cdr aexp))))))))
```
Could (+ 3 4) Yes

Or (plus 3 4) Yes

Is (plus (times 3 6) (expt 8 2)) a representation of an arithmetic expression? Yes

Try to write the function value for a new kind of arithmetic expression that is either
— a number
— a list of the atom plus followed by two arithmetic expressions
— a list of the atom times followed by two arithmetic expressions
— or, a list of the atom expt followed by two arithmetic expressions

What about

\[
\begin{align*}
&\text{(define value} \\
&(\text{lambda} (aexp) \\
&(\text{cond}) \\
&(\quad ((\text{number?} \ aexp) \ aexp)) \\
&(\quad ((\text{eq?} \ (\text{car} \ aexp) \ (\text{quote} \ plus)) \\
&(\quad \quad (+ \ (\text{value} \ (\text{cdr} \ aexp)))) \\
&(\quad \quad (\text{value} \ (\text{cdr} \ (\text{cdr} \ aexp)))))) \\
&(\quad ((\text{eq?} \ (\text{car} \ aexp) \ (\text{quote} \ times)) \\
&(\quad \quad (\times \ (\text{value} \ (\text{cdr} \ aexp)))) \\
&(\quad \quad (\text{value} \ (\text{cdr} \ (\text{cdr} \ aexp)))))) \\
&(\quad (t \ (+ \ (\text{value} \ (\text{cdr} \ aexp)))) \\
&(\quad \quad (\text{value} \ (\text{cdr} \ (\text{cdr} \ aexp))))))
\end{align*}
\]

You guessed it It's wrong

Let's try an example (plus 1 3)

(number? aexp), where 
aexp is (plus 1 3) No
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>And now recur</td>
<td>Yes</td>
</tr>
<tr>
<td>What is (cdr aexp), where aexp is (plus 1 3)</td>
<td>(1 3)</td>
</tr>
<tr>
<td>(1 3) is not our representation of an arithmetic expression</td>
<td>No, we violated The Eighth Commandment (1 3) is not a subpart that is a representation of an arithmetic expression! We obviously recurred on a list. But remember, not all lists are representations of arithmetic expressions. We have to recur on subexpressions.</td>
</tr>
<tr>
<td>How can we get the first subexpression of a representation of an arithmetic expression?</td>
<td>By taking the car of the cdr</td>
</tr>
<tr>
<td>Is (cdr (cdr aexp)) an arithmetic expression where aexp is (plus 1 3)</td>
<td>No, the cdr of the cdr is (3), and (3) is not an arithmetic expression</td>
</tr>
<tr>
<td>Again, we were thinking of the list (plus 1 3) instead of the representation for an arithmetic expression</td>
<td>Taking the car of the cdr of the cdr gets us back on the right track</td>
</tr>
<tr>
<td>What do we mean if we say the car of the cdr of aexp?</td>
<td>The first subexpression of the representation of an arithmetic expression</td>
</tr>
<tr>
<td>Question</td>
<td>Answer</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Let's write a function 1st sub-exp for arithmetic expressions           | `(define 1st-sub-exp
   (lambda (aexp)
     (cond
       (t (car (cdr aexp)))))` |
| Why do we ask `t`?                                                      | Because the first question is also the last question                   |
| Can we get by without `(cond )` if we don’t need to ask questions?      | Yes, remember one-liners                                               |
| Write 2nd sub-exp for arithmetic expressions.                          | `(define 2nd-sub-exp
   (lambda (aexp)
     (car (cdr (cdr aexp)))))` |
| Finally, let's replace (car `aexp`) by (operator `aexp`)               | `(define operator
   (lambda (aexp)
     (car aexp)))` |
Now write `value` again

```
(define value
  (lambda (aexp)
    (cond
      ((number? aexp) aexp)
      ((eq? (operator aexp) (quote plus))
        (+ (value (1st-sub-exp aexp))
           (value (2nd-sub-exp aexp))))
      ((eq? (operator aexp) (quote times))
        (* (value (1st-sub-exp aexp))
           (value (2nd-sub-exp aexp))))
      (t (^ (value (1st-sub-exp aexp))
           (value (2nd-sub-exp aexp)))))
```

Can we use this `value` function for the first representation of arithmetic expressions in this chapter?

Yes, by changing 1st sub-exp and operator

Do it!

```
(define 1st-sub-exp
  (lambda (aexp)
    (car aexp)))
```

```
(define operator
  (lambda (aexp)
    (car (cdr aexp)))))
```

Wasn't this easy?

Yes, because we used help functions to hide the representation

---

**The Ninth Commandment**

Use help functions to abstract from representations.
Have we seen representations before?  
Yes, we just did not tell you that they were representations

For what entities have we used representations?  
Truth values, Numbers

Numbers are representations?  
Yes. For example 4 stands for the concept four. We chose that symbol because we are accustomed to arabic representations

What else could we have used?  
(( ) ( ) ( ) (  )) would have served just as well. What about ((((( )))))? How about (| V)?

Do you remember how many primitives we need for numbers?  
Four number?, zero?, add1, and sub1

Let's try another representation for numbers  
( ) is our choice
How shall we represent zero now?

How is one represented?  
(( ) )

How is two represented?  
(( ) ( ) )

Got it? What's three?  
Three is (( ) ( ) ( ))

Write a function to test for the null list

(define null?  
(lambda (s)  
  (and  
    (atom? s)  
    (eq? s (quote ( ))))))
Write a function to test for zero

\[
\text{(define zero?}
\begin{array}{l}
\text{(lambda (n)} \\
\text{(null? n))})
\end{array}
\]

Can you write `add1`?

\[
\text{(define add1}
\begin{array}{l}
\text{(lambda (n)} \\
\text{(cons (quote ( )) n))})
\end{array}
\]

What about `sub1`?

\[
\text{(define sub1}
\begin{array}{l}
\text{(lambda (n)} \\
\text{(cdr n))})
\end{array}
\]

Is this correct?

Let's see

What is \((\text{sub1 } n)\) where \(n\) is \((\)\)?

No answer, but that's fine

— Recall The Law of Cdr

Rewrite `+` using this representation.

\[
\text{(define +}
\begin{array}{l}
\text{(lambda (n m)} \\
\text{(cond}
\begin{array}{l}
\text{((zero? m) n)} \\
\text{(i (add1 (+ n (sub1 m)))))})
\end{array}
\end{array}
\]

Has the definition of `+` changed?

No, only the definitions of its help functions (i.e., `zero?`, `add1`, and `sub1`) have changed

How do we define a number in general?

A number is either zero or it is one added to a number.
How many questions do we need to ask in order to write number?  Two

What is used in the natural recursion for number?  (cdr n)

Write the function number?

(define number?
  (lambda (n)
    (cond
      ((null? n) t)
      (t (and
          (null? (car n))
          (number? (cdr n)))))))

Is (cookie) a number in our representation?  No, but you deserve one now!

Go and get one!
Or better yet, make your own.

(define cookies
  (lambda ()
    (bake
      (quote (350 degrees))
      (quote (12 minutes))
      (mix
        (quote (walnuts 1 cup))
        (quote (chocolate-chips 16 ounces))
        (mix
          (mix
            (quote (flour 2 cups))
            (quote (oatmeal 2 cups))
            (quote (salt 5 teaspoon))
            (quote (baking powder 1 teaspoon))
            (quote (baking soda 1 teaspoon)))
          (mix
            (quote (eggs 2 large))
            (quote (vanilla 1 teaspoon))
            (cream
              (quote (butter 1 cup))
              (quote (sugar 2 cups))))))))
Exercises

For these exercises,

\[ aexp1 \text{ is } (1 + (3 \times 4)) \]
\[ aexp2 \text{ is } ((3 \uparrow 4) + 5) \]
\[ aexp3 \text{ is } (3 \times (4 \times (5 \times 6))) \]
\[ aexp4 \text{ is } 5 \]
\[ l1 \text{ is } ( ) \]
\[ l2 \text{ is } (3 \uparrow (66 6)) \]
\[ lexp1 \text{ is } (\text{AND (OR } x y) y) \]
\[ lexp2 \text{ is } (\text{AND (NOT } y) (\text{OR } u v)) \]
\[ lexp3 \text{ is } (\text{OR } x y) \]
\[ lexp4 \text{ is } z \]

7.1 So far we have neglected functions that build representations for arithmetic expressions. For example, \texttt{mk+exp}

```
(define mk+exp
  (lambda (aexp1 aexp2)
    (cons aexp1
      (cons (quote +)
        (cons aexp2 ( ))))))
```

makes an arithmetic expression of the form \((aexp1 + aexp2)\), where \(aexp1\), \(aexp2\) are already arithmetic expressions. Write the corresponding functions \texttt{mk*exp} and \texttt{mk↑exp}.

The arithmetic expression \((1 + 3)\) can now be built by \((\text{mk+exp } x y)\), where \(x\) is 1 and \(y\) is 3. Show how to build \(aexp1\), \(aexp2\), and \(aexp3\).
7.2 A useful function is `aexp?` that checks whether an S-expression is the representation of an arithmetic expression. Write the function `aexp?` and test it with some of the arithmetic expressions from the chapter. Also test it with S expressions that are not arithmetic expressions.

Example: `(aexp? aexp1)` is true,
       `(aexp? aexp2)` is true,
       `(aexp? l1)` is false,
       `(aexp? l2)` is false

7.3 Write the function `count-op` that counts the operators in an arithmetic expression.

Example: `(count-op aexp1)` is 2,
         `(count-op aexp3)` is 3,
         `(count-op aexp4)` is 0.

Also write the functions `count+`, `count×`, and `count↑` that count the respective operators.

Example: `(count+ aexp1)` is 1,
         `(count× aexp1)` is 1,
         `(count↑ aexp1)` is 0

7.4 Write the function `count-numbers` that counts the numbers in an arithmetic expression.

Example: `(count-numbers aexp1)` is 3,
         `(count-numbers aexp3)` is 4,
         `(count-numbers aexp4)` is 1

7.5 Since it is inconvenient to write `(3 × (4 × (5 × 6)))` for multiplying 4 numbers, we now introduce prefix notation and allow + and × expressions to contain 2, 3, or 4 subexpressions. For example, `(+ 3 2 (× 7 8)), (× 3 4 5 6)` etc. are now legal representations. ↑-expressions are also in prefix form but are still binary.

Rewrite the functions numbered? and value for the new definition of `aexp`.

Hint: You will need functions for extracting the third and the fourth subexpression of an arithmetic expression. You will also need a function `cnt-aexp` that counts the number of arithmetic subexpressions in the list following an operator.

Example: When `aexp1` is `(+ 3 2 (× 7 8))`,
         `aexp2` is `(× 3 4 5 6),` and
         `aexp3` is `↑ aexp1 aexp2`, then
         `(cnt-aexp aexp1)` is 3,
         `(cnt-aexp aexp2)` is 4,
         `(cnt-aexp aexp3)` is 2
7 7 Write the function covered? of \( lexp \) and \( lat \) that tests whether all the variables in \( lexp \) are in \( lat \).

Example: When \( \textit{li} \) is \((x\ y\ z\ u)\), then

- \((\text{covered? } lexp1 \ li)\) is true,
- \((\text{covered? } lexp2 \ li)\) is false,
- \((\text{covered? } lexp4 \ li)\) is true.

7 8 For the evaluation of \( L \) expressions we will need an \( \textit{alist} \). An \( \textit{alist} \) for \( L \)-expressions is a list of pairs. The first component of a pair is always an atom, the second one is either the number 0 (signifying false) or 1 (signifying true). The second component is referred to as the value of the variable. Write the function lookup of \( \textit{var} \) and \( \textit{alist} \) that returns the value of the first pair in \( \textit{alist} \) whose car is \( \textit{eq?} \) to \( \textit{var} \).

Example: When \( \textit{li} \) is \(((x\ 1)\ (y\ 0))\),
- \( \textit{l2} \) is \(((u\ 1)\ (v\ 1))\),
- \( \textit{l3} \) is \(),
  - \( a \) is \( y \)
  - \( b \) is \( u \), then
  - \((\text{lookup } a \ li)\) is 0,
  - \((\text{lookup } b \ li)\) is 1,
- \((\text{lookup } a \ l3)\) has no answer.

7 9 If the list of atoms in an \( \textit{alist} \) for \( L \)-expressions contains all the variables of an \( L \)-expression \( lexp \), then \( lexp \) can be evaluated with respect to this \( \textit{alist} \). (Use the function covered? from Exercise 7 7 for the appropriate test) Write the function \( \text{Mlexp} \) of \( lexp \) and \( \textit{alist} \).

\[(\text{Mlexp } lexp \ \textit{alist})\] is true
- if \( lexp \) is a variable and its value is true, or
- if \( lexp \) is an AND-expression and both subexpressions yield true, or
- if \( lexp \) is an OR expression and one of the subexpressions yields true, or
- if \( lexp \) is a NOT-expression and the subexpression yields false

Otherwise \( \text{Mlexp} \) yields false. \( \text{Mlexp} \) has no answer if the expression is not covered by

(example \( \textit{firsts } \textit{alist} \))

Example: When \( \textit{li} \) is \(((x\ 1)\ (y\ 0)\ (z\ 0))\),
- \( \textit{l2} \) is \(((y\ 0)\ (u\ 0)\ (v\ 1))\), then
(Mexp exp1 u1) is false,
(Mexp exp2 u2) is true,
(Mexp exp4 u1) is false

Hint: You will need the function lookup from Exercise 7.8

7.10 Extend the representation of L-expressions to AND and OR several subexpressions, i.e.,
(AND x (OR u v w) y).
Rewrite the function Mexp from Exercise 7.9 for this representation
Hint: Exercise 7.5 is a similar extension of arithmetic expressions.
Friends and Relations
(set? lat), where lat is (apples peaches pears plums) t, because no atom appears more than once:

(set? lat), where lat is () t because no atom appears more than once:

Try to write set?

```
(define set?
   (lambda (lat)
     (cond
       ((null? lat) t)
       (t (cond
         ((member? (car lat) (cdr lat)) nil)
         (t (set? (cdr lat)))))))
```

Simplify set?

```
(define set?
   (lambda (lat)
     (cond
       ((null? lat) t)
       (t (set? (cdr lat))))))
```

Does this work for the example
   (apple 3 pear 4 9 apple 3 4)
   Yes, since member? is now written using equal? instead of eq?

Were you surprised to see the function member? appear in the definition of set?

You should not be, because we have written member? already, and now we can use it whenever we like.
Try to write makeset, using member?

```
(define makeset
 (lambda (lat)
   (cond
     ((null? lat) (quote ( )))
     ((member? (car lat) (cdr lat))
      (makeset (cdr lat)))
     (t (cons (car lat)
              (makeset (cdr lat))))))
```

Are you surprised to see how short this is? We hope so. But don't be afraid. It's right.

Using the definition that you just wrote, what is the result of (makeset lat), where

```
(lat is (apple peach pear peach plum apple lemon peach))
```

Try to write makeset, using multirember

```
(define makeset
 (lambda (lat)
   (cond
     ((null? lat) (quote ( )))
     (t (cons (car lat)
              (makeset
               (multirember
                (car lat) (cdr lat))))))
```

What is the result of (makeset lat) using this second definition, where

```
(lat is (apple peach pear peach plum apple lemon peach))
```

(apple peach pear plum lemon)
Can you describe in your own words how the second definition of makeset works?

Here are our words:
"Makeset saves the first atom in the lat, and then recurs, after removing all occurrences of the first atom from the rest of the lat."

Does the second makeset work for the example
(apple 3 pear 4 9 apple 3 4)

Yes, since multirember is now written using equal? instead of eq?

What is (subset? set1 set2), where
set1 is (5 chicken wings), and
set2 is (5 hamburgers
   2 pieces fried chicken and
   light duckling wings)

T, because each atom in set1 is also in set2

What is (subset? set1 set2), where
set1 is (4 pounds of horseradish), and
set2 is (four pounds chicken and
   5 ounces horseradish)

Nil

Try to write subset?

```
(define subset?
  (lambda (set1 set2)
    (cond
      ((null? set1) t)
      (t (cond
          ((member? (car set1) set2)
            (subset? (cdr set1) set2))
          (t nil))))))
```
Try to write a shorter version of subset?

\[
\text{(define subset?}
\quad \text{(lambda (set1 set2))}
\quad \text{(cond}
\quad \quad \text{((null? set1) t)}
\quad \quad \text{((member? (car set1) set2))}
\quad \quad \text{(subset? (cdr set1) set2))}
\quad \text{(t nil)))}
\]

Try to write subset? with (and )

\[
\text{(define subset?}
\quad \text{(lambda (set1 set2))}
\quad \text{(cond}
\quad \quad \text{((null? set1) t)}
\quad \quad \text{(t (and}
\quad \quad \quad \text{(member? (car set1) set2))}
\quad \quad \quad \text{(subset? (cdr set1) set2)))))}
\]

What is (eqset? set1 set2), where
set1 is (6 large chickens with wings), and
set2 is (6 chickens with large wings)

Try to write eqset?

\[
\text{(define eqset?}
\quad \text{(lambda (set1 set2))}
\quad \text{(cond}
\quad \quad \text{((subset? set1 set2))}
\quad \quad \text{(subset? set2 set1))}
\quad \text{(t nil)))}
\]

Can you write eqset? with only one cond line?

\[
\text{(define eqset?}
\quad \text{(lambda (set1 set2))}
\quad \text{(cond}
\quad \quad \text{(t (and}
\quad \quad \quad \text{(subset? set1 set2))}
\quad \quad \quad \text{(subset? set2 set1)))))}
\]
Write the one-liner

```
(define eqset?
  (lambda (set1 set2)
    (and
     (subset? set1 set2)
     (subset? set2 set1)))))
```

(intersect? set1 set2), where
set1 is (tomatoes and macaroni), and
set2 is (macaroni and cheese)
t, because at least one atom in set1 is in set2

Try to write intersect?

```
(define intersect?
  (lambda (set1 set2)
    (cond
     ((null? set1) nil)
     (t (cond
         ((member? (car set1) set2) t)
         (t (intersect? (cdr set1) set2)))))))
```

Try to write the shorter version

```
(define intersect?
  (lambda (set1 set2)
    (cond
     ((null? set1) nil)
     ((member? (car set1) set2) t)
     (t (intersect? (cdr set1) set2))))))
```
Try writing intersect? with (or )

(define intersect?
  (lambda (set1 set2)
    (cond
      (null? set1) nil
      (t (or
          (member? (car set1) set2)
          (intersect? (cdr set1) set2))))))

Look back at subset? and compare for similarities

What is (intersect set1 set2), where
set1 is (tomatoes and macaroni), and
set2 is (macaroni and cheese)

Try to write intersect

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

Rewrite intersect with
(member? (car set1) set2)
replaced by
(not (member? (car set1) set2))

(define intersect
  (lambda (set1 set2)
    (cond
      (null? set1) (quote ()))
      (not (member? (car set1) set2))
      (intersect (cdr set1) set2))
      (t (cons (car set1) (intersect (cdr set1) set2))))))
Confused?

Write out the long versions and start simplifying when they are correct

What is \((\text{union } \text{set1 } \text{set2})\), where

\text{set1} is (tomatoes and macaroni casserole),
and

\text{set2} is (macaroni and cheese)

(tomatoes casserole macaroni and cheese)

Try to write \text{union}

\[(\text{define union}
\begin{array}{l}
\quad \text{(lambda (set1 set2)}
\quad \text{(cond}
\quad \quad ((\text{null? set1}) set2)
\quad \quad ((\text{member? (car set1) set2})
\quad \quad \quad (\text{union (cdr set1) set2}))
\quad \quad (t (\text{cons (car set1)}
\quad \quad \quad \quad (\text{union (cdr set1) set2))))))
\end{array}\)
\]

What is this function?

\[(\text{define xxx}
\begin{array}{l}
\quad \text{(lambda (set1 set2)}
\quad \text{(cond}
\quad \quad ((\text{null? set1}) (quote ( )))
\quad \quad ((\text{member? (car set1) set2})
\quad \quad \quad (\text{xxx (cdr set1) set2}))
\quad \quad (t (\text{cons (car set1)}
\quad \quad \quad \quad (\text{xxx (cdr set1) set2))))))))
\end{array}\)

In our words

"It is a function which returns all the atoms in \text{set1} that are not in \text{set2}.
That is, \text{xxx} is the complement function"

What is \((\text{intersectall } l \text{ set})\), where

\text{l-set} is \(((a b c) (c a d e) (e f g h a b))\)

(a)
What is (intersectall l-set), where
l-set is ((6 pears and)
  (3 peaches and 6 peppers)
  (8 pears and 6 plums)
  (and 6 prunes with lots of apples))

Now, using whatever help functions you need, write intersectall assuming that the
list of sets is non-empty

(define intersectall
  (lambda (l-set)
    (cond
      ((null? (cdr l-set)) (car l-set))
      (t (intersect (car l-set)
                    (intersectall (cdr l-set)))))))

Is this a pair?
  (pear pear) Yes, because it is a list with only two atoms

Is this a pair?
  (3 7) Yes

Is this a pair?
  (2 pair) Yes

Is this a pair?
  (full house) Yes

How can you refer to the first atom of a pair?
  By taking the car of the pair

How can you refer to the second atom of a pair?
  By taking the car of the cdr of the pair
How can you make a pair with two atoms? You cons the first atom onto the cons of the second atom onto ( ). That is, (cons a1 (cons a2 (quote ( )))).

They will be used to make representations of pairs and to get hold of parts of representations of pairs.

See Chapter 7

They will be used to improve readability as you will soon see

Redefine first, second, and build as one-liners

Does the definition of build require atoms as arguments?

What possible uses do these three functions have?

Can you write third as a one-liner?

(define third
  (lambda (l)
    (car (cdr (cdr l)))))

Is l a rel, where

l is (apples peaches pumpkin pie)

No, since l is not a list of pairs. We use rel to stand for relation
<table>
<thead>
<tr>
<th><strong>Is ( l ) a rel, where</strong></th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l ) is ((apples peaches) (pumpkin pie))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Is ( l ) a rel, where</strong></th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l ) is ((4 3) (4 2) (7 6) (6 2) (3 4))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Is ( rel ) a fun, where</strong></th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rel ) is ((4 3) (4 2) (7 6) (6 2) (3 4))</td>
<td>We use fun to stand for function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>What is (fun? ( rel )), where</strong></th>
<th>t, because (firsts ( rel )) is a set</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rel ) is ((8 3) (4 2) (7 6) (6 2) (3 4))</td>
<td>—See Chapter 3.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Try to write fun?</strong></th>
<th>How about this?</th>
</tr>
</thead>
</table>
|                     | \[
| (define fun?        | (lambda (rel)   |
| (lambda (rel)       |   (cond         |
|                      |     ((null? rel) t) |
|                      |     ((member*    |
|                      |       ((first (car rel)) (cdr rel)) |
|                      |       nil) |
|                      |     (t (fun? (cdr rel)))))))) |

<table>
<thead>
<tr>
<th><strong>When will this definition of fun? work?</strong></th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>(not (intersect? (firsts ( rel )) (second ( rel ))))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Try again to write (fun? ( rel )) so it will work</strong></th>
<th>(define fun?</th>
</tr>
</thead>
<tbody>
<tr>
<td>for the case where ( rel ) is ((8 3) (4 2) (7 6) (6 2) (3 4))</td>
<td>(lambda (rel)</td>
</tr>
<tr>
<td></td>
<td>(cond</td>
</tr>
<tr>
<td></td>
<td>((null? rel) t)</td>
</tr>
<tr>
<td></td>
<td>((member?</td>
</tr>
<tr>
<td></td>
<td>((first (car rel)) (firsts (cdr rel)))</td>
</tr>
<tr>
<td></td>
<td>nil)</td>
</tr>
<tr>
<td></td>
<td>(t (fun? (cdr rel))))))))</td>
</tr>
</tbody>
</table>
Rewrite fun? with set?

\[
\text{(define fun?}
\begin{align*}
&\text{(lambda (rel)} \\
&\quad \text{(set? (firsts rel))}))
\end{align*}
\]

What is (revrel rel), where
\[rel\text{ is ((}8\ a\) (pumpkin pie) (got sick))\]

\[
\text{((a 8) (pie pumpkin) (sick got))}
\]

Try to write revrel

\[
\text{(define revrel}
\begin{align*}
&\text{(lambda (rel)} \\
&\quad \text{(cond} \\
&\quad \quad \text{((null? rel) (quote ( ))))} \\
&\quad \quad \text{(t (cons} \\
&\quad \quad \quad \text{(cons} \\
&\quad \quad \quad \quad \text{(car (cdr (car rel))})} \\
&\quad \quad \quad \text{(cons} \\
&\quad \quad \quad \quad \text{(car (car rel))} \\
&\quad \quad \quad \quad \text{(quote ( ))))} \\
&\quad \quad \quad \text{(revrel (cdr rel)))))
\end{align*}
\]

Would the following also be correct

\[
\text{(define revrel}
\begin{align*}
&\text{(lambda (rel)} \\
&\quad \text{(cond} \\
&\quad \quad \text{((null? rel) (quote ( ))))} \\
&\quad \quad \text{(t (cons} \\
&\quad \quad \quad \text{(cons} \\
&\quad \quad \quad \quad \text{(car (cdr (car rel))})} \\
&\quad \quad \quad \text{(cons} \\
&\quad \quad \quad \quad \text{(car (car rel))} \\
&\quad \quad \quad \quad \text{(quote ( ))))} \\
&\quad \quad \quad \text{(revrel (cdr rel)))))
\end{align*}
\]

Yes, but now do you see how representation aids readability?

Can you guess why fun is not a fullfun, where
\[fun\text{ is ((}8\ 3\) (4\ 2\) (7\ 6\) (6\ 2\) (3\ 4))}\]

fun is not a fullfun, since the 2 appears more than once as a second atom of a pair
Why is $t$ the value of (fullfun? fun), where 
fun is ((8 3) (4 8) (7 6) (6 2) (3 4)) 
Because the list (3 8 6 2 4) is a set

What is (fullfun? fun), where 
fun is ((grape raisin) 
(plum prune) 
(stewed prune))

ml

What is (fullfun? fun), where 
fun is ((grape raisin) 
(plum prune) 
(stewed grape))

t, because the list (raisin prune grape) is a set

Try to write (fullfun? fun)

(define fullfun? 
(lambda (fun) 
(set? (seconds fun))))

What is another function name for fullfun? 
one-to-one?

Can you think of a second way to write one to one?

(define one-to-one? 
(lambda (fun) 
(fun? (revrel fun))))

If one of the ways you just wrote that last function was

1. Sitting down
2. Standing up
3. Standing on your head
For these exercises,

\[ r_1 \text{ is } ((a \ b) (a \ a) (b \ b)) \]
\[ r_2 \text{ is } ((c \ c)) \]
\[ r_3 \text{ is } ((a \ c) (b \ c)) \]
\[ r_4 \text{ is } ((a \ b) (b \ a)) \]
\[ f_1 \text{ is } ((a \ 1) (b \ 2) (c \ 2) (d \ 1)) \]
\[ f_2 \text{ is } () \]
\[ f_3 \text{ is } ((a \ 2) (b \ 1)) \]
\[ f_4 \text{ is } ((1 \ $) (3 \ *)) \]
\[ d_1 \text{ is } (a \ b) \]
\[ d_2 \text{ is } (c \ d) \]
\[ x \text{ is } a \]

**8.1** Write the function domset of \( rel \) which makes a set of all the atoms in \( rel \). This set is referred to as *domain of discourse* of the relation \( rel \).

Example (domset \( r_1 \)) is \( (a \ b) \),
\[ \text{(domset } r_2 \text{)} \text{ is } (c) \]
\[ \text{(domset } r_3 \text{)} \text{ is } (a \ b \ c) \]

Also write the function idrel of \( s \) which makes a relation of all pairs of the form \( (d \ d) \) where \( d \) is an atom of the set \( s \). (idrel \( s \)) is called the *identity relation on* \( s \)

Example: (idrel \( d_1 \)) is \( ((a \ a) (b \ b)) \),
\[ \text{(idrel } d_2 \text{)} \text{ is } ((c \ c) (d \ d)) \]
\[ \text{(idrel } f_2 \text{)} \text{ is } () \]

**8.2** Write the function reflexive? which tests whether a relation is *reflexive*. A relation is reflexive if it contains all pairs of the form \( (d \ d) \) where \( d \) is an element of its domain of discourse (see Exercise 8.1).

Example (reflexive? \( r_1 \)) is true,
\[ \text{(reflexive? } r_2 \text{)} \text{ is true,} \]
\[ \text{(reflexive? } r_3 \text{)} \text{ is false} \]
8.3 Write the function symmetric? which tests whether a relation is symmetric. A relation is symmetric if it is eqset? to its revrel.

Example (symmetric? r1) is false,
(symmetric? r2) is true,
(symmetric? r3) is true

Also write the function antisymmetric? which tests whether a relation is antisymmetric. A relation is antisymmetric if the intersection of the relation with its revrel is a subset of the identity relation on its domain of discourse (see Exercise 8.1).

Example (antisymmetric r1) is true,
(antisymmetric r2) is true,
(antisymmetric r4) is false

And finally, this is the function asymmetric? which tests whether a relation is asymmetric.

```
(define asymmetric? 
  (lambda (rel) 
    (null? (intersect rel (revrel rel))))))
```

Find out which of the sample relations is asymmetric. Characterize asymmetry in one sentence.

8.4 Write the function fapply of f and x which returns the value of f at place x. That is, it returns the second of the pair whose first is eq? to x.

Example: (fapply f1 x) is 1
(fapply f2 x) has no answer,
(fapply f3 x) is 2

8.5 Write the function fcomp of f and g which composes two functions. If g, contains an element (x y) and f contains an element (y z), then the composed function (fcomp f g) will contain (x z).

Example: (fcomp f1 f4) is ( ),
(fcomp f1 f3) is ( ),
(fcomp f4 f1) is ((a $) (d $)),
(fcomp f4 f3) is ((b $))

Hint: The function fapply from Exercise 8.4 may be useful.

8.6 Write the function rapply of rel and x which returns the value set of rel at place x. The value set is the set of second components of all the pairs whose first component is eq? to x.

Example (rapply f1 x) is (1),
(rapply r1 x) is (b a),
(rapply f2 x) is ( )
8.7 Write the function \texttt{run} of \( x \) and \texttt{set} which produces a relation of pairs \((x \ d)\) where \( d \) is an element of \( \texttt{set} \)

Example

\begin{align*}
\texttt{(run } x \ d1) \texttt{ is (}) (a \ a) (a \ b)),
\texttt{(run } x \ d2) \texttt{ is (}} (a \ c) (a \ d)),
\texttt{(run } x \ f2) \texttt{ is (}}
\end{align*}

8.8 Relations can be composed with the following function

\begin{verbatim}
(define rcomp
 (lambda (rel1 rel2)
   (cond
    ((null? rel1) (quote ()))
    (t (union
        (run
          (first (car rel1))
          (rapply rel2 (second (car rel1))))
        (rcomp (cdr rel1) rel2))))))
\end{verbatim}

See Exercises 8.6 and 8.7

Find the values of \((\texttt{rcomp } r1 \ r3)\), \((\texttt{rcomp } r1 \ f1)\), and \((\texttt{rcomp } r1 \ r1)\)

8.9 Write the function \texttt{transitive?} which tests whether a relation is transitive. A relation \texttt{rel} is transitive if the composition of \texttt{rel} with \texttt{rel} is a subset of \texttt{rel} (see Exercise 8.3)

Example

\begin{align*}
\texttt{(transitive? } r1 \texttt{) is true,}
\texttt{(transitive? } r3 \texttt{) is true,}
\texttt{(transitive? } f1 \texttt{) is true}
\end{align*}

Find a relation for which \texttt{transitive?} yields false

8.10 Write the functions \texttt{quasi-order?}, \texttt{partial-order?}, and \texttt{equivalence?} which test whether a relation is a \texttt{quasi-order}, a \texttt{partial-order}, or an \texttt{equivalence relation}, respectively. A relation is

\begin{itemize}
  \item quasi-order if it is reflexive and transitive,
  \item partial-order if it is a quasi-order and antisymmetric,
  \item equivalence relation if it is a quasi order and symmetric
\end{itemize}

See Exercises 8.2, 8.3, and 8.9
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remember what we did in rember and insertl at the end of Chapter 6?</td>
<td>We replaced eq? by equal?</td>
</tr>
<tr>
<td>Can you write a function rember-f tht would use either eq? or equal?</td>
<td>No, because we have not yet told you how to do this</td>
</tr>
<tr>
<td>How can you make rember remove the first a from (b c a)</td>
<td>By passing a and (b c a) as arguments to rember</td>
</tr>
<tr>
<td>How can you make rember remove the first c from (b c a)</td>
<td>By passing c and (b c a) as arguments to rember</td>
</tr>
<tr>
<td>How can you make rember f use equal? instead of eq?</td>
<td>By passing equal? as an argument to rember-f.</td>
</tr>
<tr>
<td>What is (rember f test? a l), where test? is =,¹ a is 5, and l is (6 2 3)</td>
<td>(6 2 3)</td>
</tr>
</tbody>
</table>

¹ L (setq = (function =)) or (setq #)  
Now try (rember-f = 5 '(6 2 5 3)).

What is (rember f test? a l), where test? is eq?,¹  
a is jelly, and  
l is (jelly beans are good)  
(beans are good)
And what is \((\text{rember-f} \ test^? \ a \ l)\), where 
\(test^?\) is equal?, 
a is \((\text{pop corn})\), and 
l is \((\text{lemonade} \ (\text{pop corn}) \ \text{and} \ (\text{cake}))\)

Try to write \(\text{rember-f}\)

\[
\begin{align*}
(\text{define} \ \text{rember-f} \\
(\lambda (\text{test}^? \ a \ l) \\
(\text{cond} \\
((\text{null?} \ l) (\text{quote} ( ))) \\
(t (\text{cond} \\
((\text{test}^? (\text{car} \ l) \ a)^1 \ (\text{cdr} \ l)) \\
(t (\text{cons} (\text{car} \ l) \\
(\text{rember-f} \\
(\text{test}^? \ a \ (\text{cdr} \ l)))))))))) \\
\end{align*}
\]

This is good!

\[\text{L:} (\text{funcaII} \ \text{test}^? \ (\text{car} \ l) \ a) \ \text{Use funcaII when invoking a function argument.}\]

What about the short version?

\[
(\text{define} \ \text{rember-f} \\
(\lambda (\text{test}^? \ a \ l) \\
(\text{cond} \\
((\text{null?} \ l) (\text{quote} ( ))) \\
((\text{test}^? (\text{car} \ l) \ a) (\text{cdr} \ l)) \\
(t (\text{cons} (\text{car} \ l) \\
(\text{rember-f} \ \text{test}^? \ a \ (\text{cdr} \ l))))))))
\]

How does \((\text{rember-f} \ test^? \ a \ l)\) act where 
\(test^?\) is eq?

\((\text{rember-f} \ test^? \ a \ l)\), where \(test^?\) is eq?, acts like rember

And what about \((\text{rember-f} \ test^? \ a \ l)\) where 
\(test^?\) is always equal?

This is just rember with eq? replaced by equal?
Now we have four functions which do *almost* the same thing

Yes

And rember f can simulate all the others

Yes, so let's *generate* all the other versions with rember-f.

What kind of values can functions return?

Lists, symbols, numbers t, and nil

What about functions themselves?

Yes, but you probably did not know that yet

Can you say what `(lambda (a l) )` does?

`(lambda (a l) ...)` indicates that the expression is a function that takes two arguments, a and l

Now what is

`(lambda (a)
   (lambda (x)
     (eq? x a)))`

It is a function that, when passed an argument a returns the function

`(lambda (x)
   (eq? x a))`

where a is just that argument

Using `(define )`, give the preceding function a name

```
(define eq? c
  (lambda (a)
    (lambda (x)
      (eq? x a))))
```

This is our choice
What is \((eq? c k)\) where \(k\) is salad

Its value is a function that takes \(x\) as an argument and tests whether it is \(eq?\) to salad.

So let's give it a name using \((define\ )\)

\[(define eq? salad (eq? c k))\]

where \(k\) is salad

---

1. L: (setq eq?-salad (eq?-c 'salad)).
   Use \(setq\) to define a function that will be \(fun\)called.

---

(eq? salad y), where \(y\) is tuna

---

(eq? salad y), where \(y\) is salad

---

Do we need to give a name to \(eq?\) salad

No, we may just as well ask

\[((eq?-c x) y),\]

where

\(x\) is salad, and

\(y\) is tuna

---

Now you can write a function \(re\)ember \(f\) that, when passed a function as an argument, returns a function that acts like \(re\)ember \(f\) where \(test\)\(2\) is just that argument

\[(define re\text{ember-}f\]

\[\text{lambda} (test?)\]

\[\text{lambda} (a l)\]

\[\text{cond}\]

\[((null? l) (quote ( )))\]

\[\text{quote}\]

\[\text{quote}\]

\[((test? (car l) a) (cdr l))\]

\[t (cons (car l) \text{---} )\]

\[]\\

is \text{again} a good try
Describe in your own words the result of
(rember-f test?),
where
test is eq?

A function that takes two arguments, a
and l. It compares the elements of the list
with a, and the first one that is eq? to a is
removed.

Give a name to the function which is re-
turned by
(rember f test?),
where
test is eq?

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

where
test is eq?

What is (rember-eq? a l), where
a is tuna, and
l is (tuna salad is good)

(salad is good)

No, we could have written
(((rember-f test?) a l)1, where
test is eq?,
a is tuna, and
l is (tuna salad is good)

Did we need to give a name (by defining
rember-eq?) to (rember-f test?) where
test is eq?

---

1 L: (funcall (rember-f eq)
  tuna
  (tuna salad is good))

Now, complete the line
(cons (car l) ____________ )
in rember-f so that rember-f really works

(define rember-f
  (lambda (test?)
    (lambda (a l)
      (cond
        ((null? l) (quote ( )))
        ((test? (car l) a) (cdr l))
        (t (cons (car l)
          ((rember-f test?)
            a (cdr l)))))
  ))
And now transform insertL to insertL-f the same way we have transformed rember into rember f

\[
\text{(define insertL-f)}
\]
\[
\quad\text{(lambda (test?)}
\]
\[
\quad\quad\text{(lambda (new old l)}
\]
\[
\quad\quad\quad\text{(cond}
\]
\[
\quad\quad\quad\quad\text{((null? l) (quote ( )})}
\]
\[
\quad\quad\quad\quad\text{((test? (car l) old)}
\]
\[
\quad\quad\quad\quad\quad\text{(cons new (cons old (cdr l)))))}
\]
\[
\quad\quad\quad\quad\text{(t (cons (car l)}
\]
\[
\quad\quad\quad\quad\quad\text{((insertL-f test?)}
\]
\[
\quad\quad\quad\quad\quad\quad\text{new old (cdr l)))))}})
\]

And, just for the exercise, do it to insertR

\[
\text{(define insertR-f)}
\]
\[
\quad\text{(lambda (test?)}
\]
\[
\quad\quad\text{(lambda (new old l)}
\]
\[
\quad\quad\quad\text{(cond}
\]
\[
\quad\quad\quad\quad\text{((null? l) (quote ( )})}
\]
\[
\quad\quad\quad\quad\text{((test? (car l) old)}
\]
\[
\quad\quad\quad\quad\quad\text{(cons old (cons new (cdr l)))))}
\]
\[
\quad\quad\quad\quad\text{(t (cons (car l)}
\]
\[
\quad\quad\quad\quad\quad\text{((insertR-f test?)}
\]
\[
\quad\quad\quad\quad\quad\quad\text{new old (cdr l)))))}})
\]

insertR and insertL are very similar

Yes, only the middle piece is a little bit different.

Can you write a function insert g which would insert either at the left or at the right?

If you can, get yourself some coffee cake and relax! Otherwise, don’t give up. You’ll see it in a minute.
Which pieces differ?

The second lines differ from each other. In \texttt{insertl} it is:

\begin{align*}
((\text{eq?} \ (\text{car} \ \textit{l}) \ \textit{old}) \\
(\text{cons} \ \textit{new} \ (\text{cons} \ \textit{old} \ (\text{cdr} \ \textit{l}))))
\end{align*}

but in \texttt{insertr} it is:

\begin{align*}
((\text{eq?} \ (\text{car} \ \textit{l}) \ \textit{old}) \\
(\text{cons} \ \textit{old} \ (\text{cons} \ \textit{new} \ (\text{cdr} \ \textit{l}))))
\end{align*}

Put the difference in words\footnote{We say:}

"The two functions cons \textit{old} and \textit{new} in a different order onto the cdr of the list \textit{l}"

So how can we get rid of the difference?

You probably guessed it: by passing in a function which expresses the appropriate consing

Define a function \texttt{seqr} which

1. takes three arguments, and
2. conses the first argument onto the result of consing the second argument onto the third argument

\begin{verbatim}
(define seqr
  (lambda (new old l)
    (cons new (cons old l))))
\end{verbatim}

What is

\begin{verbatim}
(define seqr
  (lambda (new old l)
    (cons old (cons new l))))
\end{verbatim}

A function which

1. takes three arguments, and
2. conses the second argument onto the result of consing the first argument onto the third argument

Do you know why we wrote these functions?

Because they express what the two differing lines in \texttt{insertl} and \texttt{insertr} express
Try to write the function `insert g` of one argument `seq`, which is `insertL`, when `seq` is `seqL`, and which is `insertR`, when `seq` is `seqR`.

\[
\begin{align*}
\text{(define insert-g} \\
&\quad \text{(lambda (seq)} \\
&\quad \quad \text{(lambda (new old l)} \\
&\quad \quad \quad \text{(cond)} \\
&\quad \quad \quad \quad ((\text{null?} \ l) (\text{quote} \ () )) \\
&\quad \quad \quad \quad ((\text{eq?} \ (\text{car} \ l) \ \text{old}) \\
&\quad \quad \quad \quad \quad (\text{seq} \ \text{new} \ \text{old} \ (\text{cdr} \ l))) \\
&\quad \quad \quad \quad (t \ (\text{cons} \ (\text{car} \ l) \\
&\quad \quad \quad \quad \quad (\text{(insert-g seq)} \\
&\quad \quad \quad \quad \quad \quad \text{new old (cdr l)})))))
\end{align*}
\]

Now define `insertL` with `insert g`.

\[
\text{(define insertL} \ (\text{insert g seqL}))
\]

And `insertR`.

\[
\text{(define insertR} \ (\text{insert g seqR}))
\]

Is there something unusual about these two definitions?

Yes. Earlier we would probably have written

\[
\text{(define insertL} \ (\text{insert-g seq}))
\]

where

\[
\text{seq is seqL}
\]

and

\[
\text{(define insertR} \ (\text{insert g seq}))
\]

where

\[
\text{seq is seqR}
\]

But when you pass functions as arguments this is not necessary.

Was it necessary to give names to `seqL` and `seqR`?

Not really. We could have passed their definitions instead.
Is this better?

Yes, because you do not need to remember as many names. You can (remember func-name 'your-mind'), where func-name is seql.

Do you remember the definition of subst

Here is the definition of subst

```
(define subst
  (lambda (new old l)
    (cond
      ((null? l) (quote ( )))
      ((eq? (car l) old)
        (cons new (cdr l)))
      (t (cons (car l)
        (subst new old (cdr l))))))))
```

Does this look familiar?

Yes, it looks like insertl or insertr. Just the answer of the second cond-line is different.

Define a function like seql or seqr for subst

What do you think about

```
(define seqs
  (lambda (new old l)
    (cons new l)))
```

And now define subst using insert g

```
(define subst (insert g seqs))
```

And what do you think xxx is

```
(define xxx
  (lambda (a l)
    ((insert-g seqrem) nil a l)))
```

where

```
(define seqrem
  (lambda (new old l)
    l))
```

Surprise: It is our old friend rember!

Hint: Step through the evaluation of

```
(xxx a l),
```

where

a is sausage, and

l is (pizza with sausage and bacon)

What is the role of nil?
What you have just seen is the power of abstraction

The Tenth Commandment
Abstract functions with common structures into a single function.

Have we seen similar functions before? Yes, we have even seen functions with similar lines

Do you remember value from Chapter 7?

\[
\text{(define value} \\
\text{ (lambda (aexp))} \\
\text{ (cond} \\
\text{ ((number? aexp) aexp)} \\
\text{ ((eq? (operator aexp) (quote plus))} \\
\text{ (+ (value (1st-sub-exp aexp)))} \\
\text{ (value (2nd-sub-exp aexp)))))} \\
\text{ ((eq? (operator aexp) (quote times))} \\
\text{ (× (value (1st-sub-exp aexp)))} \\
\text{ (value (2nd-sub-exp aexp))))}) \\
\text{ (t (↑ (value (1st-sub-exp aexp)))} \\
\text{ (value (2nd-sub-exp aexp))))}))
\]

Do you see the similarities? The last three lines are the same except for the +, ×, and ↑

Can you write a function atom-to-function that

1. Takes one argument, $x$, and
2. Returns the function $+$
   - if (eq? $x$ (quote $+$)),
3. Returns the function $\times$
   - if (eq? $x$ (quote $\times$)), and
4. Returns the function $\uparrow$
   - if (eq? $x$ (quote $\uparrow$))

\[
\text{(define atom-to-function} \\
\text{ (lambda (x)} \\
\text{ (cond} \\
\text{ ((eq? x (quote $+$)) +)} \\
\text{ ((eq? x (quote $\times$)) $\times$)} \\
\text{ ((eq? x (quote $\uparrow$)) ↑)})
\]
Can you use atom-to-function to rewrite value with only two lines inside that (cond)

Of course

\[
(\text{define value} \\
(\text{lambda} (aexp) \\
(\text{cond} \\
((\text{number?} \ aexp) aexp) \\
(t \ ((\text{atom-to-function} \\
(\text{operator} aexp)) \\
(\text{value} (1st-sub-exp aexp)) \\
(\text{value} (2nd-sub-exp aexp)))))))
\]

Is this quite a bit shorter than the first version?

Yes, but that's okay. We haven't changed its meaning.

Write the functions subset? and intersect? next to each other

\[
(\text{define subset?} \\
(\text{lambda} (set1 set2) \\
(\text{cond} \\
((\text{null?} \ set1) t) \\
(t \ (\text{and} \\
\text{member? (car set1) set2}) \\
(\text{subset? (cdr set1) set2)))))))
\]

and

\[
(\text{define intersect?} \\
(\text{lambda} (set1 set2) \\
(\text{cond} \\
((\text{null?} \ set1) nil) \\
(t \ (\text{or} \\
\text{member? (car set1) set2}) \\
(\text{intersect?} \\
(cdr set1) set2))))))
\]
Agam, these functions have the same structure

So let's abstract them into a function

(set-f? logical? const)

which can generate subset? and intersect?

\[
\text{(define set-f?}
\begin{align*}
&\text{(lambda (logical? const)} \\
&\text{(lambda (set1 set2)} \\
&\text{(cond} \\
&\text{((null? set1) const)} \\
&\text{(t (logical?)} \\
&\text{(member? (car set1) set2)} \\
&\text{((set-f? logical? const)} \\
&\text{(cdr set1) set2)))))})
\end{align*}
\]

Now, define subset? and intersect? using the function set-f?

\[
\text{(define subset? (set-f? and t))}
\]

\[
\text{(define intersect? (set-f? or ml))}
\]

almost work

Why don't they?

Because and and or are not really functions
They cannot be passed as arguments

So we write functions that do act like (and ) and (or )

Here they are

\[
\text{(define and-prime}
\begin{align*}
&\text{(lambda (x y)} \\
&\text{(and x y))})
\end{align*}
\]

\[
\text{(define or-prime}
\begin{align*}
&\text{(lambda (x y)} \\
&\text{(or x y))})
\end{align*}
\]
What does
\[(\text{and} \text{ n1 (subset? x y)})\]
do, where
- \(x\) is (red wine tastes good), and
- \(y\) is (it goes well with brie cheese)

It returns \text{n1 without} ever asking the second question!

What does
\[(\text{or-prime t (intersect? x y)})\]
do, where
- \(x\) is (red wine tastes good), and
- \(y\) is (it goes well with brie cheese)

It evaluates both questions. The first one to \(t\), the second one to \text{n1}, and then it returns \(t\)

What would (\text{or }) have done instead?

It would have answered \(t\) \text{ without} asking the second question

Why are both (\text{and }) and (\text{or }) unusual?

They do not always ask the second question

\[1 \text{ Because of this property neither (and ...) nor (or ...) can be defined as functions in terms of (cond ...), but both (and ...) and (or ...) can be expressed in terms of (cond ...):}
\[(\text{and } \alpha \beta) = (\text{cond } (\alpha \beta) (\text{t n1}))\]
and
\[(\text{or } \alpha \beta) = (\text{cond } (\alpha \text{ t}) (\text{t } \beta))\]

Macros are a mechanism for expressing these relationships.

Which values do we need to ask the question
\[(\text{or } x (\text{intersect? (cdr set1) set2}))\],
where \(x\) is the result of
\[(\text{member? (car set1) set2})\]

Only \text{set1} and \text{set2}. The rest can be reconstructed
Now write
  or prime for intersect?,
  and
  and prime for subset?

\[
\begin{align*}
  \text{(define } \text{or-prime (lambda (x set1 set2) (or x (intersect? (cdr set1) set2))))} \\
  \text{(define } \text{and-prime (lambda (x set1 set2) (and x (subset? (cdr set1) set2))))}
\end{align*}
\]

Rewrite set-f? so that it can generate subset? and intersect? with and-prime and or-prime

\[
\begin{align*}
  \text{(define set-f? (lambda (log val? const) (lambda (set1 set2) (cond ((null? set1) const) (t (log val? (member? (car set1) set2) set1 set2))))))}
\end{align*}
\]

But we have not yet defined intersect? and subset?

Well, that's what we defined or-prime and and-prime for

Do it!

\[
\begin{align*}
  \text{(define intersect? (set-f? or-prime nil))} \\
  \text{(define subset? (set-f? and prime t))}
\end{align*}
\]

Didn't we need intersect? for or prime

No, we only assumed we could define it. And now we have it.
Recall the definition of multirember. Simplify multirember by removing the inner (cond ...).

```
(define multirember
  (lambda (a l)
    (cond
      ((null? l) (quote ( )))
      ((eq? (car l) a)
       (multirember a (cdr l)))
      (t (cons (car l)
               (multirember a (cdr l)))))))
```

What is

\[(\text{multirember } (\text{quote curry} ) l)\]

where

\[l \text{ is } (a \ b \ c \ \text{curry e curry g curry})\]

This is an application where \(l\) is associated with the value

\[(a \ b \ c \ \text{curry e curry g curry})\]

It has the value

\[(a \ b \ c \ e \ g)\]

If we wrap this application by

\[(\text{lambda } (l) \ldots)\]

what do we create?

We create a function

```
(llambda (l)
  (multirember (quote curry) l))
```

We define the new function, and give it a name

```
(define Mmember-curry
  (lambda (l)
    (multirember (quote curry) l)))
```

What is

\[(\text{Mmember-curry}
  (\text{quote } (a \ b \ c \ \text{curry e curry g curry}))\]
Rewrite Member curry using three questions

\[
\begin{align*}
&\text{(define Member curry} \\
&\quad \text{(lambda } (l) \\
&\quad \quad \text{(cond} \\
&\quad \quad \quad \text{((null? } l) \text{ (quote } ())) \\
&\quad \quad \quad \text{((eq? (car } l) \text{ (quote curry))} \\
&\quad \quad \quad \text{(Member-curry (cdr } l)\text{))} \\
&\quad \quad \quad \text{(t (cons (car } l) \\
&\quad \quad \quad \quad \text{(Member curry (cdr } l)))\text{)))})
\end{align*}
\]

Compare curry maker to insert g

\[
\begin{align*}
&(\text{define curry maker} \\
&\quad \text{(lambda } (future) \\
&\quad \quad \text{(lambda } (l) \\
&\quad \quad\quad \text{(cond} \\
&\quad \quad\quad \quad \text{((null? } l) \text{ (quote } ())) \\
&\quad \quad\quad \quad \text{((eq? (car } l) \text{ (quote curry))} \\
&\quad \quad\quad \quad \text{(curry maker } future\text{) (cdr } l)\text{))} \\
&\quad \quad\quad \quad \text{(t (cons (car } l) \\
&\quad \quad\quad \quad \quad \text{(curry-maker future) \\
&\quad \quad\quad \quad \quad \quad \text{(cdr } l)))\text{)))})
\end{align*}
\]

The function curry maker is like the function Member-curry in the same way that insert-g is like insertL. It takes one extra argument future. When it is applied to an argument, it returns a function that looks like Member-curry except for the applications (curry-maker future)

Does curry maker ever use the argument future

No, unlike seq, future is just passed around when curry-maker reaches the end of the list, future is not used

Can curry maker then make Member curry

Yes, it can

Define Member curry using curry maker

\[
(\text{define Member curry } (\text{curry maker } 0))
\]

Does it matter what we use to define Member-curry

No, future is never used
Can we use curry-maker to define Mmember curry with curry-maker

Of course,

\[
\text{(define Mmember-curry}
\text{(curry maker curry maker))}
\]

If we define Mmember curry this way what does \textit{future} become?

The value of \textit{future} is curry maker

But can’t we then just use future to replace curry maker in curry maker

Yes, we sure can

We call the function we just described “function maker” because its results are functions

Write function maker

\[
\text{(define function maker}
\text{(lambda (future)}
\text{(lambda (l)}
\text{(cond}
\text{((null? l) (quote ( )}))
\text{((eq? (car l) (quote curry))}
\text{((future future) (cdr l))))}
\text{((future future) (cdr l))))}
\]

Describe in your own words the function function maker

Here is what we say:

“When the function function-maker is applied to one argument that is a function and that returns Mmember-curry when applied to one argument, then function-maker yields Mmember-curry.”

That explanation sounds as if function-maker needs an argument that is just like function maker in order to construct Mmember-curry.

Yes, that is exactly what it says
Write `mmember-curry` using just function maker

\[
\text{(define mmember-curry (function-maker function maker))}
\]

Try studying the function with \(\text{(a b c curry e curry g h curry i)}\)

If we define `mmember-curry` this way what does `future` become?

The value of `future` is function maker

Why does this definition of `mmember-curry` work?

Because the value of \((\text{future future})\) is the same as \((\text{function-maker function-maker})\) which is the same as `mmember-curry`

Do we have to define (or give a name to) `function-maker`?

No, because function maker does not appear within its definition

Do we have to associate a name with `mmember-curry` using `define` ...

No, because `mmember-curry` does not appear within its definition

True or false \(\text{no recursive function needs to be given a name with (define )}\)

True. We chose `mmember-curry` as an arbitrary recursive function

True or false \(\text{instances of add1 can be replaced by (lambda (x) (add1 x))}\)

True, because \((\text{lambda (x) (add1 x)}) n\) is \(n + 1\)

True or false: instances of \((\text{lambda (x) (add1 x))}\) can be replaced by \((\text{lambda (y) ((lambda (x) (add1 x)) y))}\)

True, because adding the extra wrapping has no effect.
True or false: instances of
\[(\text{lambda} (x) (\text{add1} x))\]
can be replaced by
\[(\text{lambda} (x)
    \quad ((\text{lambda} (x) (\text{add1} x)) x))\]

True, because in general for any function \(f\) of
one argument, \(f\) can be replaced by
\[(\text{lambda} (x) (f x)).\]
Can you think of an \(f\) where this is false?

Is the definition below the same as the
function-maker we defined earlier?

\[
(\text{define} \ \text{function-maker}
    \quad (\text{lambda} (\text{future})
        \quad (\text{lambda} (l)
            \quad (\text{cond}
                \quad ((\text{null?} \ l) (\text{quote} ()))
                \quad ((\text{eq?} (\text{car} l) (\text{quote curry}))
                    \quad ((\text{lambda} (\text{arg})
                        \quad ((\text{future future}) \text{ arg}))
                        \quad (\text{cdr} l))))
                \quad (t (\text{cons} (\text{car} l)
                        \quad ((\text{lambda} (\text{arg})
                            \quad ((\text{future future}) \text{ arg}))
                            \quad (\text{cdr} l))))))
        \quad t (\text{cons} (\text{car} l)
            \quad ((\text{lambda} (\text{arg})
                \quad ((\text{future future}) \text{ arg}))
                \quad (\text{cdr} l)))))
    \quad t (\text{cons} (\text{car} l)
        \quad ((\text{lambda} (\text{arg})
            \quad ((\text{future future}) \text{ arg}))
            \quad (\text{cdr} l)))))
\]

Yes, because for an arbitrary function \(f\) we
can always replace it by
\[(\text{lambda} (x) (f x)).\]
In our case \(f\) is the expression
\[(\text{future future}),\]
and
\(x\) is \(\text{arg}\).
Is the definition below the same as the function maker we just defined?

\[
\begin{aligned}
(\text{define function-maker} \\
(\text{lambda (future)} \\
(\text{lambda (recfun)} \\
(\text{lambda (l)} \\
(\text{cond} \\
((\text{null? } l) (\text{quote } () ) ) \\
((\text{eq? } (\text{car } l) \\
(\text{quote curry}) \\
(recfun (\text{cdr } l)))) \\
(\text{t} (\text{cons} (\text{car } l) \\
(recfun (\text{cdr } l)))))))) \\
\text{(lambda (arg) } \\
(\text{((future future) arg)))))))) \\
\end{aligned}
\]

Yes, because the atom \( l \) does not appear in \( (\text{lambda } (\text{arg}) \\
(\text{((future future) arg})) \) 

Hence, we can abstract out this piece, replacing it by an atom that is associated with it. We chose the atom \( \text{recfun} \)

Can you make the definition of function maker simpler by breaking it up into two functions?

Hint: look at the inner box

\[
\begin{aligned}
(\text{define function-maker} \\
(\text{lambda (future)} \\
(M (\text{lambda (arg)} \\
(\text{((future future) arg))))))) \\
\end{aligned}
\]

\[
\begin{aligned}
(\text{define } M \\
(\text{lambda (recfun)} \\
(\text{lambda (l)} \\
(\text{cond} \\
((\text{null? } l) (\text{quote } () ) ) \\
((\text{eq? } (\text{car } l) (\text{quote curry})) \\
(recfun (\text{cdr } l)))) \\
(\text{t} (\text{cons} (\text{car } l) \\
(recfun (\text{cdr } l)))))))) \\
\end{aligned}
\]
Why is it safe to name
\( \text{lambda (recfun)} \)

Because all the variables are explicit arguments to \( M \), or they are primitives

Write \text{member-curry} without using \text{function-maker}.

Hint: Use the most recent definition of \text{function maker} in two different places

From
\[
(\text{define member curry}
\quad (\text{function-maker function maker}))
\]

we get
\[
(\text{define member curry}
\quad ((\text{lambda (future)}
\quad \quad (M (\text{lambda (arg)}
\quad \quad \quad ((\text{future future) arg})))))
\quad (\text{lambda (future)}
\quad \quad (M (\text{lambda (arg)}
\quad \quad \quad ((\text{future future) arg)))))
\]

Do you need a rest?

Yes? Then take one

Abstract the definition of \text{member-curry} by abstracting away the association with \( M \).

Hint: wrap a \( (\text{lambda (M)}) \) around the definition

We call this function \( Y \)
\[
(\text{define Y}
\quad (\text{lambda (M)}
\quad \quad ((\text{lambda (future)}
\quad \quad \quad (M (\text{lambda (arg)}
\quad \quad \quad \quad \quad \quad \quad ((\text{future future) arg})))))
\quad \quad (\text{lambda (future)}
\quad \quad \quad (M (\text{lambda (arg)}
\quad \quad \quad \quad \quad \quad \quad ((\text{future future) arg)))))
\]

Write \text{member-curry} using \( Y \) and \( M \)
\[
(\text{define member curry (Y M))}
\]
You have just worked through the derivation of a function called "the applicative-order Y-combinator." The interesting aspect of Y is that it produces recursive definitions without the bother of requiring that the functions be named with `(define )`. Define L so that length is

```
(define L
  (lambda (recfun)
    (lambda (l)
      (cond
        ((null? l) 0)
        (t (add1 (recfun (cdr l))))))))
```

```
(define length (Y L))
```

Describe in your own words what f should be for `(Y f)` to work as expected.

Our words

"f is a function which we want to be recursive, except that the atom `recfun` replaces the recursive call, and the whole expression is wrapped in

```
(lambda (recfun ) )
```

Write length using Y, but not L, by substituting the definition for L.

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

Does the Y-combinator need to be named with `(define )` No
Rewrite length without using either Y or L

\[
\text{(define length}
  \text{\((\text{lambda } (M)\))}
  \text{\((\text{lambda } (\text{future})\))}
  \text{\((\text{lambda } (\text{arg})\))}
  \text{\((\text{future } \text{future } \text{arg}))\))}
  \text{\((\text{lambda } (\text{future})\))}
  \text{\((\text{lambda } (\text{arg})\))}
  \text{\((\text{future } \text{future } \text{arg}))\))}
  \text{\((\text{lambda } (\text{recfun})\))}
  \text{\((\text{lambda } (l)\))}
  \text{\((\text{cond})\)}
  \text{\((\text{null? } l) 0)\)}
  \text{\((t \text{ (add1 (recfun (cdr l))))))\))}
\]\n
We observe that length does not need to be named with \text{(define \ldots)}. Write an application that corresponds to
\[
\text{(length (quote (a b c)))}
\]
without using length.

\[
\text{((lambda } (M)\))
  \text{\((\text{lambda } (\text{future})\))}
  \text{\((\text{lambda } (\text{arg})\))}
  \text{\((\text{future } \text{future } \text{arg}))\))}
  \text{\((\text{lambda } (\text{future})\))}
  \text{\((\text{lambda } (\text{arg})\))}
  \text{\((\text{future } \text{future } \text{arg}))\))}
  \text{\((\text{lambda } (\text{recfun})\))}
  \text{\((\text{lambda } (l)\))}
  \text{\((\text{cond})\)}
  \text{\((\text{null? } l) 0)\)}
  \text{\((t \text{ (add1 (recfun (cdr l))))))\))}
\]

Whew, names may not be necessary, but they sure can be useful!

Does your hat still fit?

Perhaps not, if your mind has been stretched.
And when your mind has returned, enjoy yourself with a great dinner:

(escargots garlic)
(chicken Provençal)
(red wine) and Brie)

is our advice †
Look up the functions `firsts` and `seconds` in Chapter 3. They can be generalized to a function map of `f` and `l` that applies `f` to every element in `l` and builds a new list with the resulting values. Write the function `map` then write the function `firsts` and `seconds` using `map`.

Write the function `assq sf` of `a`, `l`, `sk`, and `fk`. The function searches through `l` which is a list of pairs until it finds a pair whose first component is `eq?` to `a`. Then the function invokes the function `sk` with this pair. If the search fails, `(fk a)` is invoked.

Example: When `a` is apple,

- `b1` is `()`,
- `b2` is `((apple 1) (plum 2))`,
- `b3` is `((peach 3))`,
- `sk` is `(lambda (p)
  (build (first p) (add1 (second p))))`,
- `fk` is `(lambda (name)
  (cons
    name
    (quote (not in list)))))`, then

- `(assq sf a b1 sk fk)` is `(apple not-in-list)`,
- `(assq sf a b2 sk fk)` is `(apple 2)`,
- `(assq sf a b3 sk fk)` is `(apple not-in-list)`.
9.3 In the chapter we have derived a Y combinator that allows us to write recursive functions of one argument without using define. Here is the Y-combinator for functions of two arguments:

```
(define Y2
  (lambda (M)
    ((lambda (future)
        (M (lambda (arg1 arg2)
            ((future future) arg1 arg2))))
     (lambda (future)
       (M (lambda (arg1 arg2)
            ((future future) arg1 arg2))))))))
```

Write the functions =, rempick, and pick from Chapter 4 using Y2.

Note: There is a version of (lambda ...) for defining a function of an arbitrary number of arguments, and an apply function for applying such a function to a list of arguments. With this you can write a single Y-combinator for all functions.

9.4 With the Y combinator we can reduce the number of arguments on which a function has to recur. For example, member can be rewritten as:

```
(define member Y
  (lambda (a l)
    ((Y (lambda (recfun)
          (lambda (l)
            (cond
              ((null? l) nil)
              (t (or
                  (eq? (car l) a)
                  (recfun (cdr l)))))))))
    l)))
```

Step through the application (member-Y a l) where a is x and l is (y x). Rewrite the functions rember, insertr, and subst2 from Chapter 3 in a similar manner.
In Exercises 6.7 through 6.10 we saw how to use the accumulator technique. Instead of accumulators, continuation functions are sometimes used. These functions abstract what needs to be done to complete an application. For example, multisubst can be defined as

```
(define multisubst-k
  (lambda (new old lat k)
    (cond
      ((null? lat) (k (quote ( ))))
      ((eq? (car lat) old)
       (multisubst-k new old (cdr lat)
         (lambda (d)
           (k (cons new d))))))
      (t (multisubst-k new old (cdr lat)
         (lambda (d)
           (k (cons (car lat) d)))))))))
```

The initial continuation function $k$ is always the function $(lambda (x) x)$ Step through the application of

```
(multisubst-k new old lat k),
```

where

- `new` is $y$,
- `old` is $x$, and
- `lat` is $(u v x x y z x)$

Compare the steps to the application of multisubst to the same arguments. Write down the things you have to do when you return from a recursive application, and, next to it, write down the corresponding continuation function.

In Chapter 4 and Exercise 4.2 you wrote addvec and multvec. Abstract the two functions into a single function `accum`. Write the functions `length` and `occur` using `accum`.

In Exercise 7.3 you wrote the four functions `count-op`, `count+`, `count-*`, and `count-‡`. Abstract them into a single function `count-op-f` which generates the corresponding functions if passed an appropriate help function.
Functions of no arguments are called *thunks*. If \( f \) is a thunk, it can be evaluated with \((f)\).

Consider the following version of \( \text{or} \) as a function:

\[
\begin{align*}
\text{(define or-func} \\
\quad \text{(lambda (or1 or2)} \\
\quad \quad \text{(or (or1) (or2)))})
\end{align*}
\]

Assuming that \( \text{or1} \) and \( \text{or2} \) are always thunks, convince yourself that \((\text{or } \text{or1} \text{ or2})\) and \( \text{or-func} \) are equivalent. Consider as an example

\[
\text{(or (null? l) (atom? (car l)))}
\]

and the corresponding application

\[
\text{(or-func} \\
\quad \text{(lambda ( ) (null? l))} \\
\quad \text{(lambda ( ) (atom? (car l)))})
\]

where

\[
l \text{ is ( )}
\]

Write set-f? to take or-func and and-func. Write the functions intersec? and subset? with this set-f? function.

When you build a pair with an S expression and a thunk (see Exercise 9.8) you get a stream. There are two functions defined on streams: first\$ and second\$.

Note: In practice, you can actually cons an S expression directly onto a function. We prefer to stay with the less general cons function.

\[
\begin{align*}
\text{(define first\$ first) &} \\
\text{(define second\$} \\
\quad \text{(lambda (str)} \\
\quad \quad \text{(second str)))})
\end{align*}
\]

An example of a stream is \((\text{build 1 (lambda ( ) 2)})\). Let's call this stream \( s \). \((\text{first\$ s})\) is then 1, and \((\text{second\$ s})\) is 2. Streams are interesting because they can be used to represent unbounded collections such as the integers. Consider the following definitions:

Str-maker is a function that takes a number \( n \) and a function next and produces a stream:

\[
\begin{align*}
\text{(define str-maker} \\
\quad \text{(lambda (next n)} \\
\quad \quad \text{(build n (lambda ( ) (str-maker next (next n))))})
\end{align*}
\]

With str-maker we can now define the stream of all integers like this:

\[
\text{(define int (str-maker add1 0))}
\]
Or we can define the stream of all even numbers.

(define even (str maker (lambda (n) (+ 2 n)) 0))

With the function frontier we can obtain a finite piece of a stream in a list

(define frontier
  (lambda (str n)
    (cond
      ((zero? n) (quote ()))
      (t (cons (first$ str) (frontier (second$ str) (sub1 n))))))))

What is (frontier int 10)? (frontier int 100)? (frontier even 23)?
Define the stream of odd numbers.

9.10 This exercise builds on the results of Exercise 9.9 Consider the following fun

(define Q
  (lambda (str n)
    (cond
      ((zero? (remainder (first$ str) n))
       (Q (second$ str) n))
      (t (build (first$ str)
                (lambda ()
                  (Q (second$ str) n))))))))

(define P
  (lambda (str)
    (build (first$ str) (lambda () (P (Q str (first$ str)))))))

They can be used to construct streams. What is the result of

(frontier (P (second$ (second$ int))) 10)?

What is this stream of numbers? (See Exercise 4.9 for the definition of remainder)
What is the Value of All of This?
An entry is a pair of lists whose first list is a set. Also, the two lists must be of equal length. Make up some examples for entries.

Here are our examples.

((appetizer entrée beverage)
  (paté boeuf vin))

and

((beverage dessert)
  ((food is) (number one with us)))

How can we build an entry from a set of names and a list of values?

```lisp
(define new-entry build)
```

Try to build our examples with this function.

What is `(lookup-in entry name entry)`, where

- `name` is entrée, and
- `entry` is `((appetizer entree beverage)
  (food tastes good))`

What if `name` is dessert

In this case we would like to leave the decision about what to do with the user of lookup-in-entry.

How can we accomplish this?

Lookup-in-entry will take an additional argument which is a help function that is invoked when `name` is not found in the first list of an entry.

How many arguments do you think this extra function should take?

We think it should take one `name` Why?
Here is our definition of lookup in entry

```
(define lookup-in-entry
  (lambda (name entry entry-f)
    (lookup-in entry-help
      name
      (first entry)
      (second entry)
      entry-f))))
```

Write the help function

```
(define lookup-in-entry-help
  (lambda (name names values entry f)
    (cond
      ((null? names) (entry-f name))
      ((eq? (car names) name)
        (car values))
      (t (lookup-in entry help
        name
        (cdr names)
        (cdr values)
        entry-f))))))
```

A table (also called an \textit{environment}) is a list of entries. Here is one example: ( ), the empty table. Make up some others.

Here is one

```
(((appetizer entrée beverage)
  (paté boeuf vin))
((beverage dessert)
  ((food is) (number one with us))))
```

The function \texttt{extend-table} takes an entry and a table (possibly the empty one) and creates a new table by putting the new entry in front of the old table. Define the function \texttt{extend-table}

```
(define extend table cons)
```

What is

```
(lookup-in table name table table f)
```
where

\texttt{name} is entrée,
\texttt{table} is (((appetizer dessert)
  (spaghetti spumoni))
  ((appetizer entrée beverage)
    (food tastes good))), and
\texttt{table-f} is (lambda (name) ..

It could be either spaghetti or tastes, but we will have lookup-in-table search the list of entries in order. So it is spaghetti.
Write lookup-in-table.
Hint: don't forget to get some help

\[
\begin{align*}
\text{define lookup-in-table} & \quad (\text{lambda} \ (\text{name} \ \text{table} \ \text{table} \ f)) \\
& \quad (\text{cond} \\
& \quad \quad ((\text{null? table}) (\text{table} f \ \text{name})) \\
& \quad \quad (t (\text{lookup-in entry} \ \text{name} \ (\text{car table}) \ (\text{lambda} \ (\text{name}) \\
& \quad \quad (\text{lookup-in-table} \ \text{name} \ (\text{cdr table}) \ \text{table-f}))))))
\end{align*}
\]

Can you describe what the following function represents:
\[
(\text{lambda} \ (\text{name}) \\
(\text{lookup-in table} \ \text{name} \ (\text{cdr table}) \ \text{table-f}))
\]

This function is the action to take when the name is not found in the first entry

In the Preface we mentioned that sans serif font would be used to represent data. Up to this point it has hardly ever mattered. From this point on until the end of the book you must be very conscious of whether or not a particular symbol is in sans serif.

Remember to be very conscious as to whether or not a symbol is in sans serif

Did you notice that "sans serif" was not in sans serif?
We hope so This is "sans serif" in sans serif.

Have we chosen a good representation for programs?
Yes They are all S-expressions so they can be data for functions

What kind of functions?
For example, value
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you remember value from Chapter 7?</td>
<td>Recall that value is the function that returns the natural value of expressions.</td>
</tr>
<tr>
<td>What is the value of (car (quote (a b c)))</td>
<td>a</td>
</tr>
<tr>
<td>What is (value e), where e is (car (quote (a b c)))</td>
<td>a</td>
</tr>
<tr>
<td>What is (value e), where e is (quote (car (quote (a b c))))</td>
<td>(car (quote (a b c)))</td>
</tr>
<tr>
<td>What is (value e), where e is (add1 6)</td>
<td>7</td>
</tr>
<tr>
<td>What is (value e) where e is 6</td>
<td>6, because numbers are self evaluating</td>
</tr>
<tr>
<td>What is (value e) where e is nothing</td>
<td>nothing has no value</td>
</tr>
<tr>
<td>What is (value e) where e is ((lambda (nothing) (cons nothing (quote ( )))) (quote (from nothing comes something)))</td>
<td>((from nothing comes something))</td>
</tr>
<tr>
<td>Question</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>What is the type of e where e is 6</td>
<td>*self-evaluating</td>
</tr>
<tr>
<td>What is the type of e where e is nil</td>
<td>*identifier</td>
</tr>
<tr>
<td>What is the type of e where e is cons</td>
<td>*identifier</td>
</tr>
<tr>
<td>What is (value e) where e is car</td>
<td>(primitive car)</td>
</tr>
<tr>
<td>What is the type of e where e is nothing</td>
<td>*identifier</td>
</tr>
<tr>
<td>What is the type of e where e is (lambda (x y) (cons x y))</td>
<td>*lambda</td>
</tr>
<tr>
<td>What is the type of e where e is ((lambda (nothing)</td>
<td>*application</td>
</tr>
<tr>
<td>(cond (nothing (quote something)) (t (quote nothing))) nil)</td>
<td></td>
</tr>
</tbody>
</table>

How many types do you think there are? We found six
*self-evaluating,
*quote,
*identifier,
*lambda,
*cond, and
*application
If actions are functions that do "the right thing" when applied to the appropriate type of expression, what should value do?

You guessed it. It would have to find out the type of expression it was passed and then use the associated action.

Do you remember atom-to-function from Chapter 9?

We found atom-to-function useful when we rewrote value for numbered expressions.

Below is a program that produces the correct action (or function) for each possible S expression.

```
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (t (list-to-action e)))))
```

Define the help function atom-to-action.

```
(define atom-to-action
  (lambda (e)
    (cond
      ((number? e) *self evaluating)
      (t *identifier))))
```

1. Ill-formed S expressions such as (quote a b) and (lambda e) are not considered here. They can be detected by an appropriate function to which S-expressions are submitted before they are passed on to the interpreter.

Now define the help function list-to-action.

```
(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 cond)) *cond)
          (t *application))))
      (t *application))))
```
Assuming that expression-to action works, we can use it to define value and meaning.

\[
\text{(define value}
\begin{align*}
\text{(lambda } (e) \\
\text{(meaning } e \text{ (quote ( ))}))
\end{align*}
\text{)}
\]

It is the empty table. The function value, together with all the functions it uses, is called an interpreter.

\[
\text{(define meaning}
\begin{align*}
\text{(lambda } (e \text{ table}) \\
\text{((expression-to-action } e \text{) } e \text{ table}))
\end{align*}
\text{)}
\]

What is (quote ( )) in the definition of value?

How many arguments should actions take according to the above?

Two, the expression \( e \) and a table which is initially ( ).

Here is the action for self-evaluating expressions.

\[
\text{(define *self-evaluating}
\begin{align*}
\text{(lambda } (e \text{ table}) \\
\text{e})
\end{align*}
\text{)}
\]

Yes, it just returns that expression, and this is all we have to do for 0, 1, 2, .

Is it correct?

Here is the action for *quote

\[
\text{(define *quote}
\begin{align*}
\text{(lambda } (e \text{ table}) \\
\text{(text-of-quotatation } e))
\end{align*}
\text{)}
\]

(define text-of-quotatation second)

Define the help function text-of-quotatation.
Given that the table contains the values of identifiers, write the action \texttt{*identifier}.

\begin{verbatim}
(define *identifier
  (lambda (e table)
    (lookup-in-table
      e table initial-table)))
\end{verbatim}

Here is initial table

\begin{verbatim}
(define initial-table
  (lambda (name)
    (cond
      ((eq? name (quote t)) t)
      ((eq? name (quote nil)) nil)
      (t (build
        (quote primitive)
        name)))))
\end{verbatim}

It handles cases that are not in \texttt{table}. We defined it so that it gives values to predetermined identifiers like \texttt{t, nil, cons, zero?}, etc

When is it used?

What is the value of \texttt{(lambda (x) x)}

We don't know yet, but we know that it must be the representation of a non-primitive function

How are non-primitive functions different from primitives?

We know what primitives do; non-primitives are defined by their arguments and their function bodies

So when we want to use a non primitive we need to remember its formal arguments and its function body

At least. Fortunately this is just the \texttt{cdr} of a lambda-expression

And what else do we need to remember?

We will also put in the table in case we need it later
Here is the action *lambda

```
(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
      (cons table (cdr e))))))
```

What is (meaning e table), where e is (lambda (x) (cons x y)), and table is (((y z) ((8) 9)))

It is probably a good idea to define some help functions for getting back the parts in this three element list (i.e., the table, the formal arguments, and the body) Write table-of, formals-of, and body-of

Describe (cond ) in your own words

It is a special form which takes a list of cond-lines. It considers each line in turn. If the question part on the left is false, then it looks at the rest of the lines. Otherwise it proceeds to answer the right part

Here is the function evcon which does what we just said in words

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

Write the help functions question-of and answer-of.
Now use the function evcon to write the action *cond

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

```
(define cond-lines cdr)
```

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aren't these help functions useful?</td>
<td>Yes, they make things quite a bit more readable  But you already knew this.</td>
</tr>
<tr>
<td>Are you now familiar with the definition of *cond?</td>
<td>Probably not</td>
</tr>
<tr>
<td>How can you become familiar with it?</td>
<td>The best way is to try an example  A good one is:</td>
</tr>
<tr>
<td></td>
<td>(*cond e table),</td>
</tr>
<tr>
<td></td>
<td>where</td>
</tr>
<tr>
<td></td>
<td>e is (cond (coffee klatsch) (t party)), and</td>
</tr>
<tr>
<td></td>
<td>table is (((coffee)</td>
</tr>
<tr>
<td></td>
<td>(t))</td>
</tr>
<tr>
<td></td>
<td>(((klatsch party)</td>
</tr>
<tr>
<td></td>
<td>(5 (6))))</td>
</tr>
<tr>
<td>Have we seen how the table gets used?</td>
<td>Yes, *lambda and *identifier use it.</td>
</tr>
<tr>
<td>But how do the identifiers get into the table?</td>
<td>In the only action we have not defined, *application</td>
</tr>
<tr>
<td>How is an application represented?</td>
<td>An application is a list of expressions whose car position contains an expression whose value is a function.</td>
</tr>
</tbody>
</table>
How does an application differ from a special form, like `(and )`, `(or )`, or `(cond )`?

An application must always determine the meaning of all its arguments.

Before we can apply a function do we have to get the meaning of all arguments?

Yes

Write a function `evls` which takes a list of (representations of) arguments and a table, and returns a list composed of the meaning of each argument.

```
(define evls
  (lambda (args table)
    (cond
      ((null? args) (quote ()))
      (t (cons (meaning (car args) table)
               (evls (cdr args) table)))))))
```

What else do we need before we can determine the meaning of an application?

We need to find out what its function-of means.

And what then?

Then we apply the meaning of the function to the meaning of the arguments.

Here is the function *application*

```
(define *application
  (lambda (e table)
    (apply
      (meaning (function-of e) table)
      (evls (arguments-of e) table))))
```

Is it correct?

Of course. We just have to define `apply`, `function-of`, and `arguments-of` correctly.

Write function of and arguments-of

```
(define function-of car)
(define arguments-of cdr)
```
How many different kinds of functions are there?

Two primitives and non primitives

What are the two representations of functions?

(primitive primitive-name) and:
(non-primitive (table formals body))
The list (table formals body) without the non-primitive tag is called a closure.

Write primitive? and non primitive?

(define primitive?
  (lambda (l)
    (eq?
      (first l)
      (quote primitive))))

(define non-primitive?
  (lambda (l)
    (eq?
      (first l)
      (quote non-primitive))))

Now we can write the function apply

Here it is

(define apply1
  (lambda (fun vals)
    (cond
      ((primitive? fun)
        (apply-primitive
          (second fun) vals))
      ((non-primitive? fun)
        (apply-closure
          (second fun) vals)))))
This is the definition of apply primitive:

```
(define apply-primitive
  (lambda (name vals)
    (cond
      ((eq? name (quote car))
       (car (first vals)))
      ((eq? name (quote cdr))
       (cdr vals))
      ((eq? name (quote cadr))
       (car (first vals)))
      ((eq? name (quote cdar))
       (cdr (first vals)))
      ((eq? name (quote cons))
       (cons (first vals) (second vals)))
      ((eq? name (quote eq?))
       (eq? (first vals) (second vals)))
      ((eq? name (quote atom?))
       (atom? vals))
      ((eq? name (quote null?))
       (null? (first vals)))
      ((eq? name (quote number?))
       (number? (first vals)))
      ((eq? name (quote zero?))
       (zero? (first vals)))
      ((eq? name (quote add1))
       (add1 (first vals)))
      ((eq? name (quote sub1))
       (sub1 (first vals)))))
```

1 In apply-primitive the interpreter could check for applications of cdr to () or sub1 to 0, etc.

Fill in the blanks

Is apply closure the only function left?

Yes, and apply-closure must be the part that extends the table.

How could we find the result of \((f \ a \ b)\), where

\(f\) is \((\text{lambda} \ (x \ y) \ (\text{cons} \ x \ y))\)
\(a\) is 1, and
\(b\) is (2)

That's tricky. But we know what to do to find the meaning of

\((\text{cons} \ x \ y)\)

where

\text{table} is \(((x \ y) \ (1 \ (2))))
Why can we do this?

Here, we don’t need apply closure

Can you generalize the last two steps?

Applying a non primitive function to a list of values is the same as finding the meaning of the associated closure’s body with its table extended by an entry of the form

\[(formals\ values)\]

\[formals\ is\ the\ formals\ of\ the\ associated\ closure\ and\ values\ is\ the\ result\ of\ evals\]

Have you followed all this?

If not, here is the definition of apply-closure

\[
(define\ apply-closure
  (lambda\ (closure\ vals)
    (meaning\ (body\ of\ closure)
      (extend-table
        (new-entry
          (formals-of\ closure\ vals)
          (table-of\ closure))))))
\]

This is a complicated function and it deserves an example.

In the following

\[closure\ is\ (((((u\ v\ w)
  (1\ 2\ 3))
  ((x\ y\ z)
    (4\ 5\ 6)))
  (x\ y)
  (cons\ z\ x))\]

and

\[vals\ is\ (((a\ b\ c)\ (d\ e\ f)))\]

What will be the new arguments for meaning?

The new \(e\) for meaning will be \((cons\ z\ x)\) and the new \(table\) for meaning will be

\[(((x\ y)
  ((a\ b\ c)\ (d\ e\ f)))
  (((u\ v\ w)
    (1\ 2\ 3))
  ((x\ y\ z)
    (4\ 5\ 6))))\]
What is the meaning of \((\text{cons } z \ x)\) where
\[
z \text{ is } 6, \text{ and } \\
x \text{ is } (a \ b \ c)
\]
The same as
\[
(\text{meaning } e \ \text{table}) \\
\text{where} \\
e \text{ is } (\text{cons } z \ x), \text{ and} \\
\text{table is } (((x \ y) \\
(a \ b \ c) \ (d \ e \ f)) \\
(u \ v \ w) \\
(1 \ 2 \ 3) \\
(x \ y \ z) \\
(4 \ 5 \ 6)))
\]

Let's find the meaning of all the arguments
What is
\[
(\text{evlis } \text{args } \text{table}) \\
\text{where} \\
\text{args is } (z \ x), \\
\text{and} \\
\text{table is } (((x \ y) \\
(a \ b \ c) \ (d \ e \ f)) \\
(u \ v \ w) \\
(1 \ 2 \ 3) \\
(x \ y \ z) \\
(4 \ 5 \ 6)))
\]
In order to do this we must find both
\[
(\text{meaning } e \ \text{table}) \\
\text{where} \\
e \text{ is } z, \\
\text{and} \\
(\text{meaning } e \ \text{table}) \\
\text{where} \\
e \text{ is } x
\]

What is the \((\text{meaning } e \ \text{table})\) where
\[
e \text{ is } z
\]
6, by using *identifier

What is \((\text{meaning } e \ \text{table})\) where
\[
e \text{ is } x
\]
(a b c), by using *identifier

So, what is the result of evlis
\[
(6 \ (a \ b \ c)), \text{ because evlis returns a list of the meanings.}
\]
We are now ready to \(\text{apply } \text{fun vals}\) where
\[
\text{fun is (primitive cons), and}
\]
\[
\text{vals is (6 (a b c)).}
\]
Which path will we take?

Which condition is chosen for
\[
(\text{apply-primitive name vals})
\]
where
\[
\text{name is cons, and}
\]
\[
\text{vals is (6 (a b c))}
\]

The third:
\[
((\text{eq? name (quote cons))})
\]
\[
(\text{cons (first vals) (second vals))})
\]

What is (first vals) where
\[
\text{vals is (6 (a b c))}
\]

\[
6
\]

What is (second vals) where
\[
\text{vals is (6 (a b c))}
\]

\[
(a b c)
\]

What is (cons alpha beta) where
\[
\text{alpha is 6, and}
\]
\[
\text{beta is (a b c)}
\]

\[
(6 a b c)
\]

What is
\[
((\text{lambda } (u v))
\]
\[
(\text{lambda } (b))
\]
\[
(\text{cond}
\]
\[
(b u)
\]
\[
(t v)))))
\]
\[
alpha
\]
\[
beta)
\]
where

It is a shadow of the list (6 a b c)
Why?

Because we can define cons by

\[
\text{(define cons}
\begin{align*}
&\quad \text{(lambda} (u \ v) \\
&\quad \text{(lambda} (b) \\
&\quad \text{(cond} \\
&\quad \quad (b \ u) \\
&\quad \quad (t \ v))))
\end{align*}
\]

How does this work?

Well, let's step through a simple example.

\[
\text{(define lunch (cons x y))}
\]

The function lunch takes an argument, \(b\). If \(b\) is true, the car, \(x\), is returned (i.e., apple). If \(b\) is false, the cdr, \(y\), is returned (i.e., ( )

Define car and cdr for lists using this representation.

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

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

What is \(\text{(car lunch)}\)

apple

What is \(\text{(cdr lunch)}\)

( )

Is that what we wanted?

Yes

Can we cons lunch onto lunch?

Yes, (cons lunch lunch)

What is the Value of All of This?
Chapter 7 we showed that numbers could be represented with lists. Can you recall when not and null were defined?

But what about (define )

It isn't needed either because recursion can be obtained with the Y combinator.

Does that mean we can run the interpreter on the interpreter if we do the transformation with the Y combinator?

Yes, but don't bother

Does that mean we can run the interpreter on the interpreter if we do the transformation with the Y combinator?

Yes, it's time for a banquet

"Koot's Banquet"

Is this

\[
(Y \ (\lambda \ (\infty) \\
\quad (\text{cons} \ (\text{sub}1 \ 1) \ \infty)))
\]

the same as this

\[
(Y \ (\lambda \ (\infty) \\
\quad (\text{cons} \ 0 \ \infty)))
\]

Try it with

\[
\begin{align*}
\text{(define cons} \\
\quad (\lambda \ (u \ v) \\
\quad (\lambda \ (b) \\
\quad (\text{cond} \\
\quad (b \ u) \\
\quad (t \ v)))))
\end{align*}
\]
For these exercises,

\[ e1 \text{ is } ((\lambda x) \text{ (cond (atom? x) (quote done)) (null? x) (quote almost)) (t (quote never))) (quote ——)), \]

\[ e2 \text{ is } (((\lambda x y) (\lambda u) \text{ (cond (u x) (t y)))) 1 (()) nil), \]

\[ e3 \text{ is } ((\lambda x) ((\lambda x) (\lambda x) (add1 x)) (add1 4))) 6), \]

\[ e4 \text{ is } (3 (quote a) (quote b)), \]

\[ e5 \text{ is } (\lambda (l) \text{ (cons (quote lat) l)}), \]

\[ e6 \text{ is } (\lambda (l) \text{ (l) a (quote b))} \]

10.1 Make up examples for \( e \) and step through \( (\text{value } e) \). The examples should include values, numbers, and quoted S-expressions.

10.2 Make up some S-expressions, plug them into the _____ of \( e1 \), and step through the application of \( (\text{value } e1) \).

10.3 Step through the application of \( (\text{value } e2) \). How many closures are produced during the application?

What is the Value of All of This?
10.4 Consider the expression $e3$. What do you expect to be the value of $e3$? Which of the three $x$'s are “related”? Verify your answers by stepping through (value $e3$) Observe to which $x$ we add one

10.5 Design a representation for closures and primitives such that the tags (i.e., primitive and non-primitive) at the beginning of the lists become unnecessary. Rewrite the functions that are knowledgeable of the structures. Step through (value $e3$) with the new interpreter

10.6 Just as the table for predetermined identifiers initial-table, all tables in our interpreter can be represented as functions. Then, the function extend-table is changed to

```lisp
(define extend-table
  (lambda (entry table)
    (lambda (name)
      (cond
        ((member? name (first entry))
         (pick (index name (first entry)))
         (second entry)))
        (t (table name))))))
```

(For pick see Chapter 4; for index see Exercise 4.5.) What else has to be changed to make the interpreter work? Make the least number of changes. Make up an application of value to your favorite expression and step through it to make sure you understand the new representation. Hint: Look at all the places where tables are used to find out where changes have to be made

10.7 Write the function *lambda?, which checks whether an S expression is really a representation of a lambda-function.

Example: (*lambda? e5) is true,

(*lambda? e6) is false,

(*lambda? e2) is false.

Also write the functions *quote? and *cond? which do the same for quote- and cond expressions

10.8 Non-primitive functions are represented by lists in our interpreter. An alternative is to use functions to represent functions. For this we change *lambda to

```lisp
(define *lambda
  (lambda (e table)
    (build
      (quote non-primitive)
      (lambda (vals)
        (meaning (body of e)
          (extend-table
            (new-entry (formals of e) vals)
            table))))))
```
How do we have to change apply-closure to make this representation work? Do we need to change anything else? Walk through the application (value e2) to become familiar with this new representation.

10.9 Primitive functions are built repeatedly while finding the value of an expression. To see this, step through the application (value e3) and count how often the primitive for add1 is built. However, consider the following table for predetermined identifiers:

```
(define initial-table
  ((lambda (add1)
      (lambda (name)
        (cond
          ((eq? name (quote t)) t)
          ((eq? name (quote nil)) nil)
          ((eq? name (quote add1)) add1)
          (t (build (quote primitive) name))))))

(build (quote primitive) add1)))
```

Using this initial-table, how does the count change? Generalize this approach to include all primitives.

10.10 In Exercise 2.4 we introduced the (if ...) form. We saw that (if ...) and (cond ...) are interchangeable. If we replace the function *cond by *if where:

```
(define *if
  (lambda (e table)
    (if (meaning (test-pt e) table)
        (meaning (then-pt e) table)
        (meaning (else-pt e) table))))
```

we can almost evaluate functions containing (if ...) What other changes do we have to make? Make the changes. Take all the examples from this chapter that contain a (cond ...), rewrite them with (if ...), and step through the modified interpreter. Do the same for e1 and e2.
You have reached the end of your introduction to Lisp and recursion. Are you now ready to tackle a major programming problem in Lisp? Programming in Lisp requires two kinds of knowledge: understanding the nature of symbolic programming and recursion, and discovering the lexicon, features, and idiosyncrasies of a particular Lisp implementation. The first of these is the more difficult intellectual task. If you understand the material in this book, you have mastered that challenge. In any case, it would be well worth your time to develop a fuller understanding of all the capabilities in Lisp—this requires getting access to a running Lisp system and mastering those idiosyncrasies. If you want to understand Lisp in greater depth, the first, second, and fourth references are good choices for further reading. Abelson, Sussman, and Sussman [1] develops the concepts required for building large programs. Dybvig [2] describes Scheme, the Lisp-descendant used throughout this book. Steele [4] is the reference manual for Common Lisp, an increasingly popular dialect. Reading these books will give you some of the flavor of the features found in complete Lisp systems. We recommend Suppes [5] to the reader who wants to explore symbolic manipulation in a non-programming context, and Hofstadter [3] to the reader who wants to examine the place of recursion in the context of human thought.

References


+, 62, 127
-, 63
×, 67
↑, 77
<, 76
>, 75
=, 76.
1st-sub-exp, 124.
2nd-sub-exp, 124
L, 172
M, 170
P, 179.
Q, 179.
Y, 171
Y2, 176
add1, 61, 127
addvec, 67.
all-nuns, 80
and, 106
and-prime, 164
answer-of, 189.
*application, 191
apply, 192.
apply-closure, 194.
apply-primitive, 193
arguments-of, 191.
asymmetric?, 148
atom?, 10.
atom-to-action, 186
atom-to-function, 160
body-of, 189.
build, 143
car, 5, 197
cdr, 6, 197
cond, 17
*cond, 190
cond-lunes, 190
cons, 8, 197, 198
cookies, 129
define, 17
eq?, 12.
eqan?, 80
eq?-c, 153
eqlist?, 107, 108
eq?-salad, 154
eqset?, 138, 139
equal?, 108.
evcon, 189
evlis, 191.
expression-to-action, 186
extend-table, 182, 200.
first, 143.
first$, 178
firsts, 48
formals-of, 189
frontier, 179
f*, 112.
fullfun?, 146
fun?, 145
function-maker, 167
function-of, 191
g*, 112.
*identifier, 188
*if, 201
initial-table, 188, 201
insert-g, 158.
insertL, 56, 158
insertL-f, 156
insertL*, 104, 110
insertR, 54, 158.
insertR-f, 156
insertR*, 100
intersect, 140.
intersect?, 139, 164
intersectall, 142
lambda, 17.
*lambda, 189, 200
lat?, 17.
leftmost, 98
length, 77, 172
lst-to-action, 186
lookup-in-entry, 182
lookup-in-entry-help, 182
lookup-in-table, 183.
lunch, 197
makeset, 136
meaning, 187.
member?, 24.
member2?, 35.
member-Y, 176.
member*, 104.
mk+exp, 130.
multiinsertL, 90.
multiinsertR, 89
multirember, 86
multisubst, 91
multisubst-k, 177
new-entry, 181
non-atom?, 98
non-primitive?, 192
no-nms, 79
not, 98
null?, 9, 126.
number?, 79, 128
numbered?, 118
occur, 91
occur*, 102
one?, 92.
one-to-one?, 146
operator, 124
or, 23
or-func, 178
or-prime 164
pick, 78
primitive?, 192
question-of, 189
quote, 115.
*quote, 187
remainder, 83
rember, 38, 108
rember-eq?, 155.
rember-f, 152, 155
rember*, 99
rempick, 79, 92
revrel, 145
second, 143
second$, 178
*self-evaluating, 187
seqrem, 159
seqL, 157.
seqR, 157
seqS, 159
set?, 135
set-f?, 164
str-maker, 178.
sub1, 61, 127.
subset?, 137, 164
subst, 56, 159
subst2, 57.
subst*, 103.
table-of, 189
mrember-curry, 165
rcomp, 149.
text-of-quotiation, 187
thrd, 143
union, 141
value, 125, 161, 187
vec+, 73
xxx, 141, 159
zaro?, 62, 127
The Five Laws

The Law of Car
Car is defined only for non null lists

The Law of Cdr
Cdr is defined only for non-null lists
The cdr of any non-null list is always another list

The Law of Cons
Cons takes two arguments
The second argument of cons must be a list
The result is a list

The Law of Null?
Null? is defined only for lists

The Law of Eq?
Eq? takes two arguments
Each must be an atom.